

**UNIVERSIDADE FEDERAL DE SANTA CATARINA  
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA  
EM AUTOMAÇÃO E SISTEMAS**

Jim Lau

**Uma Estratégia para Implementação de Tolerância a Intrusões  
em Redes WAN**

Florianópolis

2014



Jim Lau

**Uma Estratégia para Implementação de Tolerância a Intrusões  
em Redes WAN**

Tese submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas para a obtenção do Grau de Doutor em Engenharia de Automação e Sistemas.

Orientador: Prof. Dr. Joni da Silva Fraga

Florianópolis

2014

Ficha de identificação da obra elaborada pelo autor,  
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Lau, Jim

Uma Estratégia para Implementação de Tolerância a  
Intrusões em Redes WAN / Jim Lau ; orientador, Joni da  
Silva Fraga - Florianópolis, SC, 2014.  
216 p.

Tese (doutorado) - Universidade Federal de Santa  
Catarina, Centro Tecnológico. Programa de Pós-Graduação em  
Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de Automação e Sistemas. 2. Tolerância a  
Intrusões. 3. Composição de Serviços Web. 4. Virtualização.  
I. Fraga, Joni da Silva . II. Universidade Federal de  
Santa Catarina. Programa de Pós-Graduação em Engenharia de  
Automação e Sistemas. III. Título.

# UMA ESTRATÉGIA PARA IMPLEMENTAÇÃO DE TOLERÂNCIA A INTRUSÕES EM REDES WAN

Jim Lau

Tese submetida à Universidade Federal de Santa Catarina  
como parte dos requisitos para a obtenção do Grau de Doutor em  
Engenharia de Automação e Sistemas.

---

Prof. Joni da Silva Fraga, Dr.

Orientador

---

Prof. Jomi Fred Hübner, Dr.

Coordenador do Programa de Pós-Graduação em Engenharia de  
Automação e Sistemas

**Banca Examinadora:**

---

Prof. Joni da Silva Fraga, Dr.

Presidente



---

Prof. Elias Procópio Duarte Junior, Dr.

---

Prof. Fernando Luís Dotti, Dr.

---

Prof. Eduardo Adilio Pelinson Alchieri, Dr.

---

Prof. Eduardo Augusto Bezerra, Dr.

---

Prof. Carlos Barros Montez, Dr.





*Aos meus pais Lau Shing Lit  
e Lau Yeung Pik Yee.*

*Aos meus irmãos Lau Cheuk Lung  
e Lau Cheuk Lun.*



## **AGRADECIMENTOS**

Gostaria de iniciar esse trabalho agradecendo a Deus, por ter me dado a dádiva da vida, e por sempre ter me aberto as portas e me guiado pelos caminhos que escolhi traçar.

Ao professor Joni da Silva Fraga e ao professor Lau Cheuk Lung por ter acreditado em mim e em meu trabalho, aceitando como meus orientadores, sempre me auxiliando com direcionamentos e sugestões em minhas dúvidas e questionamentos.

A todas as pessoas que tive a oportunidade de conhecer no decorrer do mestrado, colegas do LCMI, professores do DAS e funcionários da PGEAS.



# UMA ESTRATÉGIA PARA IMPLEMENTAÇÃO DE TOLERÂNCIA A INTRUSÕES EM REDES WAN

**Jim Lau**

Abril/2014

Orientador: Prof. Joni da Silva Fraga, Dr.

Área de Concentração: Controle, Automação e Sistemas

**Palavras-chave:** Tolerância a Intrusões, Composição de Serviços Web, Virtualização

A Internet é conhecida por agregar os mais diversos sistemas computacionais, que variam desde a arquitetura de máquina, sistemas operacionais até aplicativos finais de usuários. Além disso, nos últimos anos existe uma demanda crescente na utilização desses sistemas computacionais em aplicações críticas que forneçam um serviço correto e ininterrupto sobre redes de longa distância como a Internet (WANs). Mas, estas WANs se caracterizam como redes públicas de fácil acesso. Isto torna estes ambientes suscetíveis a um conjunto de problemas que colocam em risco a integridade destes serviços críticos quando disponíveis nestas redes. São problemas de segurança, com ataques aos mesmos tentando explorar possíveis vulnerabilidades das implementações destes serviços. São também problemas de confiabilidade devido ao uso contínuo e concorrente nestes ambientes abertos.

Nesta tese, fizemos um estudo visando encontrar soluções que melhorassem as condições de segurança e confiabilidade destes serviços, usando tecnologias como *Web Services* e Virtualização aliadas a conceitos e soluções algorítmicas próprios de Sistemas Distribuídos. As nossas soluções desenvolvidas para estes problemas tiveram como motivação as aplicações práticas das mesmas, mas sem deixar de lado aspectos conceituais importantes de Sistemas Distribuídos. Com base nestes objetivos, introduzimos um modelo híbrido de tolerância a intrusões e faltas que provoca a

separação de faltas bizantinas e de *crash*. As faltas bizantinas são tratadas a nível local usando a virtualização. As faltas de *crash* que envolvem menores custos são tratadas a nível de sistema distribuídos.

# UMA ESTRATÉGIA PARA IMPLEMENTAÇÃO DE TOLERÂNCIA A INTRUSÕES EM REDES WAN

**Jim Lau**

April/2014

Advisor: Prof. Joni da Silva Fraga, Dr.

Area of Concentration: Control, Automation and Systems

Keywords: Intrusion Tolerance, Virtualization, Web Service Composition

The Internet is known to add all kinds of computer systems, ranging from machine architecture, operating systems, end user applications. Moreover, the last years there is an increasing demand on the use of these computing systems in critical applications that provide a correct and uninterrupted service on long distance networks such as the Internet (WAN). However, these WANs are characterized as public networks for easy access. This makes these environments susceptible to a set of problems, which put at risk the integrity of these critical services when available in these networks. They are security issues with the same attacks attempting to exploit potential vulnerabilities of these services implementations. Also are reliability problems due to continuous use and concurrent in these open environments.

In this thesis, we did a study to finding solutions that improve the safety and reliability of these services, using technologies such as Web Services and Virtualization concepts and combined with own algorithmic solutions Distributed Systems. Our solutions to these problems were developed as motivation the practical applications of the same, but without neglecting important conceptual aspects of Distributed Systems. Based on these goals, we introduce a hybrid model for intrusion tolerance and faults which causes the separation of Byzantine and crash faults. Byzantine

faults are handled locally using virtualization. The crash faults involving lower costs are treated at the level of distributed system.



## LISTA DE FIGURAS

Figura 2.1 - Classificação da semântica de faltas.....	35
Figura 2.2 – Relação entre ataque, vulnerabilidade e intrusão.....	36
Figura 2.3 – Passo de uma rodada do protocolo Paxos.....	43
Figura 2.4 – Algoritmo PBFT em funcionamento normal.....	48
Figura 2.5 – Execução do protocolo de acordo do Zyzzyva.....	51
Figura 2.6 – Operação normal do algoritmo MinBFT.....	53
Figura 2.7 – Operação normal do algoritmo EBAWA.....	59
Figura 3.1 – Interação entre as entidades da SOA.....	69
Figura 3.2 – Colaboração típica na arquitetura dos serviços <i>web</i> .....	70
Figura 3.3 – Orquestração e Coreografia.....	71
Figura 3.4 – Infraestrutura de serviços em <i>Web Services</i> [Tai, <i>et al.</i> , 2004]..	72
Figura 3.5 – Exemplo de orquestração.....	74
Figura 3.6 – Fluxo de processo e sua execução.....	76
Figura 3.7 – Orquestrações Centralizada (a) e Descentralizada (b).....	78
Figura 3.8 – Orquestração centralizada e descentralizada [Chafle, <i>et al.</i> , 2004]. .....	80
Figura 3.9 – Orquestrações: (a) Centralizada e (b) Descentralizada com <i>triggers</i> .....	82
Figura 4.1 – Modelo de interação entre os processos.....	91
Figura 4.2 – Mecanismos de memória compartilhada.....	94
Figura 4.3 – Execução do protocolo sem falha.....	96
Figura 4.4 – Execução do protocolo na presença de réplica maliciosas.....	98
Figura 4.5 – Framework de serviços em replicação ME.....	107
Figura 4.6 – Comparação da arquitetura com falhas e sem falhas.....	111
Figura 4.7 – Tempo de resposta considerando usuários simultâneos.....	112
Figura 5.1 – Visão geral do modelo.....	117
Figura 5.2 – Organização interna de um sítio.....	118
Figura 5.3 – <i>Buffers</i> na memória compartilhada.....	120
Figura 5.4 - Sítios e seus domínios de nomes.....	121

Figura 5.5 – <i>Paxos</i> com líder pré-definido. ....	123
Figura 5.6 –Protocolo <i>Paxos</i> executado em nossa proposta.....	124
Figura 5.7 – Relação entre <i>timeouts</i> no sistema proposto. ....	145
Figura 6.1 – Perda de mensagem <i>propose</i> . ....	170
Figura 6.2 – Perda completa de mensagens <i>propose</i> . ....	171
Figura 6.3 – Perda da mensagem de admissão. ....	172
Figura 6.4 – Sítio com recepções insuficientes de mensagens <i>accept<sub>k</sub>'s</i> .....	172
Figura 6.5 – Restauração de configuração mínima com exclusão de coordenador.....	173
Figura 6.6 – Camadas do protótipo. ....	176
Figura 6.7 – Comportamento do sistema sem falhas. ....	179
Figura 6.8 – Histograma dos testes sem falhas. ....	180
Figura 6.9 – Gráficos da execução com 0,1 % de instâncias com falhas. ....	182
Figura 6.10 – Histograma com 0,1 % de instâncias com falhas .....	183
Figura 6.11 – Execução com 1% de falhas no envio da mensagem <i>reqS</i> .....	184
Figura 6.12 – Histograma dos testes com 1% falha.....	184
Figura 6.13 – Execução com 50% de falhas .....	185
Figura 6.14 – Histograma dos testes com 50% de falhas.....	185
Figura 6.15 – Execução com 0,1% de Falha. ....	187
Figura 6.16 – Histograma dos testes com 0,1% de falha. ....	187
Figura 6.17 – Execução com 1% de falhas. ....	188
Figura 6.18 – Histograma dos testes com 1% de falhas.....	189
Figura 6.19 – Execução com 50% de falhas. ....	190
Figura 6.20 – Histograma dos testes com 50% de falhas.....	190
Figura 6.21 – Execução com 0,1 % de instâncias com falhas. ....	191
Figura 6.22 – Histograma dos testes com 0,1 % de instâncias com falhas...	192
Figura 6.23 – Execução com 1% de falhas. ....	193
Figura 6.24 – Histograma dos testes com 1% de falhas.....	193
Figura 6.25 – Execução com 50% de falhas. ....	194
Figura 6.26 – Histograma dos testes com 50% de falhas.....	194
Figura 6.27 – Execução com 0,1 % de instâncias com falhas. ....	195

Figura 6.28 – Histograma dos testes com 0,1 % de instâncias com falhas...	196
Figura 6.29 – Execução com 1 % de falhas.....	196
Figura 6.30 – Histograma dos testes com 1% de falha.....	197
Figura 6.31 – Execução com 50 % de falhas.....	197
Figura 6.32 – Histograma dos testes com 50% de falhas.....	198
Figura 6.33 – Tempo de resposta o aumento progressivo de falhas.....	198
Figura 6.34 – Throughput em operação sem falha e com falhas.....	199



## LISTA DE TABELA

Tabela 2.1 – Passos do protocolo Steward. ....	58
Tabela 2.2 – Comparativo entre as diferentes propostas. ....	66
Tabela 3.1 – Comparação dos trabalhos discutidos.....	87
Tabela 4.1 - Componentes internos dos engines.....	108
Tabela 6.1 – Distribuição das falhas .....	181
Tabela 6.2 - Intervalo de confiança para a execução com 0,1% de falhas ...	183
Tabela 6.3 – Intervalo de confiança para a execução com 1% de falhas ....	184
Tabela 6.4 – Intervalo de confiança para a execução com 50% de falhas....	186
Tabela 6.5 – Intervalo de confiança para a execução com 0,1% de falha. ...	188
Tabela 6.6 – Intervalo de confiança para a execução com 1% de falhas. ....	189
Tabela 6.7 – Intervalo de confiança para a execução com 50% de falhas....	191
Tabela 6.8 – Intervalo de confiança para a execução com 0,1% de falhas...	192
Tabela 6.9 – Intervalo de confiança para a execução com 1% de falhas. ....	193
Tabela 6.10 – Intervalo de confiança para a execução com 50% de falhas..	195
Tabela 6.11 - Comparativo entre as diferentes propostas. ....	203



## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b> .....	<b>25</b>
1.1	MOTIVAÇÃO .....	26
1.2	OBJETIVOS .....	28
1.3	ESTUDOS E TRABALHOS.....	29
1.4	ORGANIZAÇÃO DO TEXTO .....	30
<b>2</b>	<b>CONCEITOS DE ALGORITMOS DISTRIBUÍDOS SEGUROS</b> .	<b>33</b>
2.1	CLASSIFICAÇÃO DE FALTAS.....	33
2.2	TOLERÂNCIA A INTRUSÕES EM SISTEMAS DISTRIBUÍDOS ..	35
2.3	PROBLEMAS CLÁSSICOS DE SISTEMAS DISTRIBUÍDOS.....	37
2.3.1	<b>Consenso</b> .....	<b>37</b>
2.3.2	<b>Difusão Atômica</b> .....	<b>38</b>
2.3.3	<b>Replicação Máquina de Estado</b> .....	<b>39</b>
2.3.4	<b>Faltas Maliciosas Independentes: Diversidade de Projetos</b> .....	<b>41</b>
2.4	ESTADO DA ARTE .....	41
2.4.1	<b>Paxos</b> .....	<b>42</b>
2.4.2	<b>PBFT: Practical Byzantine Fault Tolerance</b> .....	<b>47</b>
2.4.3	<b>Zyzyva: Speculative Byzantine Fault Tolerance</b> .....	<b>50</b>
2.4.4	<b>MinBFT: Minimal Byzantine Fault Tolerance</b> .....	<b>52</b>
2.4.5	<b>Mencius: Efficient Replicated State Machine for WANs</b> .....	<b>55</b>
2.4.6	<b>Steward: Byzantine Fault-Tolerance Replication to WANs</b> .....	<b>56</b>
2.4.7	<b>EBAWA: Efficient Byzantine Agreement for WANs</b> .....	<b>58</b>
2.4.8	<b>Replicação Máquina de Estados no ZZ</b> .....	<b>60</b>
2.4.9	<b>SMIT: Replicação ME usando Virtualização</b> .....	<b>62</b>
2.5	CONSIDERAÇÕES .....	63

2.6	CONCLUSÃO DO CAPÍTULO.....	66
<b>3</b>	<b>CONCEITOS EM ARQUITETURAS ORIENTADAS A SERVIÇO.....</b>	<b>67</b>
3.1	ARQUITETURA ORIENTADA A SERVIÇOS .....	68
3.2	SERVIÇOS WEB .....	69
3.3	COMPOSIÇÃO DE SERVIÇOS.....	71
<b>3.3.1</b>	<b>Coreografia.....</b>	<b>72</b>
<b>3.3.2</b>	<b>Orquestração.....</b>	<b>73</b>
<b>3.3.3</b>	<b>A Linguagem para Orquestração – WS-BPEL.....</b>	<b>75</b>
3.4	ORQUESTRAÇÃO DESCENTRALIZADA.....	77
<b>3.4.1</b>	<b>Trabalhos sobre Orquestração Descentralizada.....</b>	<b>79</b>
3.5	TRABALHOS RELACIONADOS.....	82
<b>3.5.1</b>	<b>Trabalhos Envolvendo Serviços Web e Faltas por Crash.....</b>	<b>82</b>
<b>3.5.2</b>	<b>Composição de Serviços Web e Tolerância a Faltas por Crash</b>	<b>84</b>
<b>3.5.3</b>	<b>Trabalhos com Serviços Web e Faltas Bizantinas.....</b>	<b>85</b>
<b>3.5.4</b>	<b>Resumo dos Trabalhos Relacionados Usando Tecnologia Web Services.....</b>	<b>86</b>
3.6	CONCLUSÃO DO CAPÍTULO.....	87
<b>4</b>	<b>SERVIÇO TOLERANTE A INTRUSÕES.....</b>	<b>89</b>
4.1	MODELO DE SISTEMA .....	90
<b>4.1.1</b>	<b>Abordagem de Replicação do Modelo.....</b>	<b>92</b>
<b>4.1.2</b>	<b>Virtualização.....</b>	<b>93</b>
<b>4.1.3</b>	<b>Memória Compartilhada .....</b>	<b>94</b>
<b>4.1.4</b>	<b>Agreement Service .....</b>	<b>95</b>
4.2	PROTOCOLO <i>ITVM</i> .....	96
<b>4.2.1</b>	<b>Execução Sem-Falha do Protocolo <i>ITVM</i>.....</b>	<b>96</b>



4.2.2	<b>Execução do Protocolo ITVM na Presença de Réplicas Maliciosas</b>	97
4.2.3	<b>Descrição Detalhada do Algoritmo</b>	99
4.2.4	<b>Ativação de Novas Réplicas no AS</b>	103
4.2.5	<b>Checkpoint das Threads de Execução</b>	104
4.2.6	<b>Considerações sobre a Proposta Algorítmica Apresentada</b>	105
4.3	<b>SERVIÇOS REPLICADOS</b>	106
4.3.1	<b>Componentes e suas Interações no Framework Desenvolvido</b>	107
4.4	<b>PROTÓTIPO E TESTES</b>	110
4.5	<b>CONCLUSÃO</b>	112
5	<b>SERVIÇO TOLERANTE A INTRUSÕES EM REDES DE LONGA DISTÂNCIA</b>	115
5.1	<b>MODELO DE SISTEMA</b>	115
5.1.1	<b>Abordagem de Replicação do Modelo</b>	117
5.1.2	<b>Componentes em Nível de Sítio</b>	118
5.1.3	<b>Relação de Componentes em Nível Global</b>	120
5.2	<b>VISÃO ALGORÍTMICA DO MODELO</b>	122
5.2.1	<b>Descrição das Ações em Evolução Normal do Protocolo</b>	123
5.2.2	<b>Interações do Cliente com o Serviço Replicado Hierárquico</b>	125
5.2.3	<b>Protocolo de Admissão de Requisições de Clientes</b>	128
5.2.4	<b>Processamento de Requisições Admitidas</b>	130
5.2.5	<b>Processamento de Mensagens <i>PROPOSE</i></b>	131
5.2.6	<b>Processamento de Mensagens <i>ACCEPT</i></b>	133
5.2.7	<b>Processamento de Mensagens <i>RESPONSE</i></b>	136
5.3	<b>MECANISMOS DE CHECKPOINT</b>	141
5.4	<b>TROCAS DE VISÃO</b>	143

5.4.1	Temporizações na Primeira Parte do Protocolo .....	144
5.4.2	Temporizações na Segunda Parte do Protocolo .....	155
5.5	OPERAÇÕES DE INICIAÇÃO E REMOÇÃO DE RÉPLICAS .....	160
5.5.1	Função de Iniciação de VMs.....	161
5.5.2	Função de Remoção de VMs.....	163
5.6	CONCLUSÃO DO CAPÍTULO.....	165
6	RESULTADOS EXPERIMENTAIS .....	167
6.1	ANÁLISE DO SUPORTE ALGORÍTMICO HIERÁRQUICO.....	167
6.1.1	Propriedades do Paxos.....	167
6.1.2	Recuperação do Cliente em Instâncias de Protocolo .....	169
6.1.3	Evolução do Protocolo em Situações de Exceções.....	170
6.1.4	Custos do Protocolo Proposto .....	173
6.2	PROTÓTIPO DESENVOLVIDO.....	175
6.2.1	Detalhes de Implementação do Protótipo .....	176
6.2.2	Serviço de Aplicação com Replicação ME .....	177
6.2.3	Testes sobre o Protótipo.....	178
6.2.4	Experimentos com Execuções Normais de Protocolo.....	179
6.2.5	Experimentos com Simulação de Falhas no Protocolo.....	180
6.2.6	Comportamento com Falha no Envio de Mensagem de Admissão .....	181
6.2.7	Comportamento com Falha no Envio de Mensagem <i>Propose</i> . 186	
6.2.8	Comportamento em Falhas no Envio da Mensagem <i>Accept</i> ... 191	
6.2.9	Comportamento em Desacordo de Respostas de Réplicas Locais .....	195
6.2.10	Impacto das Falhas no Tempo Médio de Resposta.....	198

<b>6.2.11</b>	<b>Throughput.....</b>	<b>199</b>
<b>6.2.12</b>	<b>Considerações sobre o Protótipo .....</b>	<b>201</b>
6.3	COMPARAÇÕES DE NOSSAS PROPOSIÇÕES COM TRABALHOS RELACIONADOS .....	202
6.4	CONCLUSÃO DO CAPÍTULO .....	204
<b>7</b>	<b>CONCLUSÃO .....</b>	<b>205</b>
7.1	VISÃO GERAL DO TRABALHO.....	205
7.2	REVISÃO DOS OBJETIVOS.....	206
7.3	CONTRIBUIÇÕES DA TESE.....	208
7.4	TRABALHOS FUTUROS.....	208
<b>8</b>	<b>Referências Bibliográficas .....</b>	<b>211</b>







# 1 INTRODUÇÃO

A Internet agrega os mais diversos sistemas computacionais, que variam desde a arquitetura de máquina, sistemas operacionais até aplicativos finais de usuários. O sucesso de um ambiente tão diversificado se deve à definição de um conjunto de protocolos padronizados que garante a interoperabilidade entre as aplicações, independente de arquitetura de máquina ou sistema operacional entre outras características.

Na Internet, a cada dia novos serviços e novas facilidades são oferecidos aos seus usuários e desenvolvedores. O crescimento na utilização da mesma também tem sido marcado pelo surgimento de novas tecnologias. A virtualização é um exemplo destas tecnologias que vem contribuindo para a constante evolução da rede mundial. Esta tecnologia tem favorecido a portabilidade das aplicações e o surgimento de novas formas de sistemas distribuídos, onde é enfatizada a independência de máquinas físicas e a flexibilidade de reconfigurações e adaptações às necessidades de processamento. A computação em nuvem é um bom exemplo destes novos sistemas que fazem o uso das facilidades proporcionadas pela tecnologia de virtualização.

Por outro lado, a necessidade de interoperabilidade está sempre presente neste ambiente que agrega os mais diversos sistemas computacionais. Este ambiente plural da rede mundial enfatiza tecnologias integradoras. Os Serviços Web (*Web Services*) [Booth, *et al.*, 2004] cumprem este papel integrador tão necessário em redes de longa distância. Esta tecnologia implementa conceitos de Arquitetura Orientada a Serviço (*Service-Oriented Architecture - SOA*) [Zdun, *et al.*, 2006] que introduz conceitos centrados em modelo *stateless* e fraco acoplamento (em outras palavras, sem sessões de processamentos) de comunicação. Este modelo fracamente acoplado evidencia as necessidades de descobertas dinâmicas de serviço, da composição e da interoperabilidade entre serviços. A principal característica de *Web Services* é transparência em relação a plataformas, tornando-se assim ideal para a integração de aplicações em ambientes complexos.

Além disso, nos últimos anos a Internet, tem se caracterizado por uma demanda crescente na utilização em aplicações críticas que forneçam um serviço correto e ininterrupto. Entretanto, a Internet se caracteriza como uma rede pública de fácil acesso. Isto torna este ambiente suscetível a um conjunto de problemas que colocam em risco a integridade destes serviços críticos quando disponíveis nestas redes. São problemas de segurança, com ataques aos mesmos tentando explorar possíveis

vulnerabilidades das implementações destes serviços. São também problemas de confiabilidade devido ao uso contínuo e concorrente de alta demanda nestes ambientes abertos.

Entretanto, ainda existem lacunas em aberto, ligadas à disponibilidade e à segurança de aplicações distribuídas. O conjunto de especificações padronizadas pela W3C [W3C, 2005] e a OASIS [OASIS, 2005] preenchem muitos aspectos ligados à segurança, mas não contemplam requisitos de disponibilidade e confiabilidade. Este aspecto tem motivado alguns trabalhos no sentido de propor extensões aos padrões citados para adicionar mecanismos de tolerância a faltas e a intrusões nessas soluções orientadas a serviço.

## 1.1 MOTIVAÇÃO

A motivação central nos nossos trabalhos é o desenvolvimento de mecanismos de tolerância a intrusões. A justificativa maior para o seu desenvolvimento está na existência de uma crescente preocupação por infraestruturas críticas que construídas sobre tecnologias de informação, redes e serviços, sejam sempre disponíveis e seguras. Estas infraestruturas devem manter a continuidade de seus serviços, pois a menor ruptura desta continuidade pode representar sérias perdas financeiras ou de vidas humanas. A disponibilidade e a veracidade de dados coletados a partir destas infraestruturas são requisitos muito importantes para aplicações críticas. A execução de operações a partir destas infraestruturas pode provocar catástrofes se não atenderem os requisitos citados. São exemplos destas infraestruturas: *Home Bankings*, *Online Shops*, etc.

A integração de aplicações e sistemas via rede mundial (tal como, a Internet) sempre foi complexa e onerosa [Veronese, *et al.*, 2009]. É para corrigir esse cenário que surgiram tecnologias como a de *Web Services* e a virtualização que enfatizam a interoperabilidade e a integração de aplicações em diferentes ambientes computacionais. Apesar de toda a flexibilidade provida por estas tecnologias existem lacunas não devidamente preenchidas. Estas tecnologias são projetadas para ser usadas em aplicações disponíveis em *WANs*, portanto, sistemas geograficamente dispersos e de fácil acesso (redes abertas). Aplicações distribuídas que fazem uso desta tecnologia baseada em serviços, estão sempre sujeitas a uma grande variedade de ataques e intrusões.

Mecanismos que atendam aos requisitos de disponibilidade e segurança deveriam ser naturais em modelos, padrões e plataformas cuja finalidade é a programação distribuída em sistema de ambientes abertos e de larga escala. Este não é o caso destas tecnologias. Por outro lado, existem várias especificações introduzidas para tratar aspectos de



segurança, por exemplo, *XACML* [Moses, 2005], *SAML* [Ragouzis, *et al.*, 2005], *WS-Security* [Atkinson, *et al.*, 2004], *WS-Trust* [Nadalin, *et al.*, 2005], entre outras. Em geral essas especificações, que não abrangem a disponibilidade de serviços.

O principal problema encontrado na literatura para a proposição de arquiteturas que suportem a disponibilidade mesmo em ambientes de intrusões é o fato de que os custos de soluções são normalmente proibitivos em ambientes caracterizados por WANs [Veronese, *et al.*, 2010].

Foi neste contexto de orientação a serviços em ambientes de redes de longa distância que nos propusemos a investigar as necessidades e os requisitos para a proposição de uma infraestrutura com suporte a tolerância a faltas e intrusões.

A composição de serviços fornece mecanismo próprio para estes ambientes complexos. Soluções caras como a execução de um PBFT [Castro e Liskov, 1999] poderiam ser confinadas em ambientes restritos. A construção de soluções para ambientes mais amplos poderia ser facilitada usando a combinação de soluções locais. A composição de serviços é motivadora para soluções hierárquicas, mas também é pelas possibilidades de integração de negócios (aplicações), suportes e serviços para a tolerância a faltas e intrusões. O padrão mais utilizado atualmente é a *WS-BPEL* [Arkin, *et al.*, 2007]. Essa linguagem serve para descrever orquestrações através de um processo de negócio executável.

A proposta de nosso trabalho foi então centrada em um modelo de integração que pudesse agregar técnicas de tolerância a faltas bizantinas com os serviços de aplicação. Este modelo e a infraestrutura de serviços resultante deveriam estar fundamentados em composição de serviços. A infraestrutura decorrente de nossas propostas procurou respeitar os padrões existentes para a integração de algoritmos de suporte à replicação ativa (modelo Máquina de Estados, ME [Schneider, 1990]) que é a técnica normalmente usada, quando o alvo é a tolerância a faltas maliciosas.

Nas soluções dos problemas apontados, foi introduzir modelos que fizessem uso da flexibilidade fornecida pelas tecnologias citadas neste texto. Tentar novas maneiras de implementar a replicação Máquina de Estados com uso da flexibilidade destas tecnologias e ferramentas tão populares nos dias de hoje.

No entanto, não poderíamos desconsiderar os novos conceitos e propostas presentes na literatura. As tecnologias citadas, por exemplo, facilitam o uso de componentes confiáveis [Neves, *et al.*, 2004]. Este conceito reduz em custo os protocolos para Máquinas de Estados. No desenvolvimento de algoritmos tolerantes a faltas e intrusões, a inclusão

destes componentes tem sido apresentado na literatura como importante para baixar as necessidades de réplicas e o número de passos em algoritmos de acordo [Castro e Liskov, 1999] [Kotla, *et al.*, 2007] [Veronese, *et al.*, 2008] [Júnior, *et al.*, 2010](necessários para o suporte MEs). Se considerarmos ambientes de larga escala, estes ganhos não podem ser dispensados.

Na literatura, geralmente os trabalhos sobre faltas bizantinas apresentam o limite máximo de réplicas maliciosas fixo (em  $f$  falhas). Isto evidentemente permite que o protocolo se mantenha em progresso de sua execução. Mas, no entanto, as falhas que vão ocorrendo no ciclo de vida destes sistemas, reduzindo a capacidade de tolerar intrusões e falhas. Normalmente, não existe uma recuperação automática dos elementos corrompidos (faltosos) e com isto a robustez do sistema vai sendo reduzida. Além disso, quando esse limite de  $f$  falhas é alcançado durante a sua execução, qualquer outra falha subsequente no sistema faz com que o serviço deixe de funcionar corretamente. Desta maneira, é necessário que se tenha um mecanismo para o tratamento de réplicas maliciosas ou de rejuvenescimento das réplicas que atue dinamicamente no sistema e continuamente [Garcia, *et al.*, 2011]. A virtualização também pode ajudar neste quesito.

Nesta tese, a nossa motivação, portanto, sempre foi o estudo visando encontrar soluções que melhorassem as condições de segurança e confiabilidade de serviços e aplicações em WANs. As nossas soluções desenvolvidas para estes problemas tiveram como objetivo as aplicações práticas das mesmas, mas sem deixar de lado aspectos conceituais importantes de Sistemas Distribuídos.

## 1.2 OBJETIVOS

O objetivo geral que norteou nossos trabalhos foi o estudo de modelos e arquiteturas para serviços tolerantes a intrusões em ambientes de redes de longa distância. A ideia inicial era o desenvolvimento de modelos utilizando o conceito de elemento confiável que fizesse a separação de comportamentos faltosos. Com isto, produziríamos um modelo híbrido de tolerância a intrusões e faltas, separando faltas bizantinas de *crashes*. As faltas bizantinas por serem mais caras teriam seus mecanismos restritos ao nível local (de máquina servidora). O tratamento destas faltas usaria replicação MEs implementadas com o uso da virtualização. As faltas de *crash* que envolvem esse custo é tão menor que possibilita o uso em redes de larga escala. Muitos conceitos e modelos

novos propostos em sistemas distribuídos serviram de base para os nossos desenvolvimentos na concepção de um modelo híbrido e hierárquico.

Portanto, a introdução de modelos e infraestruturas de serviços que viessem a ser desenvolvidas deveria possuir a capacidade de permitir que clientes e provedores de serviços interajam de forma segura usufruindo de serviço que tenham suporte a tolerância a intrusões e garantias das propriedades de confiabilidade, disponibilidade e integridade das informações. Estes pontos formaram o foco central de nossos trabalhos.

### 1.3 ESTUDOS E TRABALHOS

Dentro da caracterização geral de nossos objetivos sobre os modelos de tolerância a intrusões e faltas para uso em grandes ambientes, vários estudos e trabalhos atendendo objetivos específicos foram consequência de nossos esforços neste período:

- Estudos de algoritmos de tolerância a faltas bizantinas (BFT) projetados para ambientes de larga escala que serviram de base para nossas proposições.
- Concretização de modelos de ME usando virtualização e concretizando os conceitos e propriedades de elemento confiável e de separação de faltas.
- Estratificação de soluções para ambientes de larga escala como *WANs* de modo a produzir protocolos hierárquicos.
- Uso de composição via várias *engines* para produzirem orquestrações descentralizadas e hierarquizadas de serviços replicados.
- Desenvolvimento de protótipos (infraestruturas) de nossos modelos de BTF que permitem de maneira flexível a construção de Máquinas de Estado tolerantes a intrusões, atendendo também requisitos de larga escala.

## 1.4 ORGANIZAÇÃO DO TEXTO

Este texto está estruturado em sete capítulos. Este capítulo inicial descreveu o contexto geral do trabalho, sua motivação e objetivos. Os demais capítulos encontram-se assim organizados:

- O capítulo 2 apresenta os conceitos de sistemas distribuídos e a literatura de tolerância a faltas bizantinas (BFT). Os conceitos apresentados são necessários para a compreensão dos diversos textos deste documento. Neste capítulo, também são apresentados as principais abordagens e modelos de BFT presentes na literatura.
- O capítulo 3 descreve conceitos e trabalhos ligados a Arquiteturas Orientadas a Serviço. É apresentada a literatura recente sobre composição de serviços.
- O capítulo 4 descreve a nossa primeira experiência com ME para a tolerância a intrusões. Neste capítulo é definido um modelo que faz uso da virtualização para tolerar e tratar réplicas de serviço corrompidas por intrusões: o modelo ITVM. A noção de elemento confiável é central neste modelo. Também neste modelo, é enfatizada a separação da execução da aplicação em réplicas da ME e do serviço de acordo necessário. O modelo de ME é todo alocado em uma máquina física servidora.
- O capítulo 5 descreve nosso protocolo hierárquico que sustenta a replicação ME em ambientes de redes de longa distância. Este protocolo faz uso do modelo definido no capítulo anterior, e estendendo o mesmo de modo a atender requisitos de disponibilidade. É a versão que chamamos de ITVM Distribuído. A separação de faltas em modelo híbrido e a possibilidade de confinar os custos de BFT em sítios torna o protocolo adequado para ambientes de longa distância. Em nível global de sistemas distribuídos o protocolo define uma necessidade menor de replicação devido à limitação de falhas neste nível a somente comportamentos de *crashes*.
- O capítulo 6 apresenta uma análise realizada sobre o protocolo e descrevemos os nossos resultados de protótipo e testes desenvolvidos para verificar o comportamento do protocolo proposto.

- Finalmente, o capítulo 7 traz nossas considerações finais sobre nossas neste trabalho.



## 2 CONCEITOS DE ALGORITMOS DISTRIBUÍDOS SEGUROS

A Internet tem sido um meio de comunicação de extrema importância para organizações e indivíduos estabelecerem e divulgarem seus serviços. Diante disto, nos últimos anos, alguns requisitos de serviços passaram a ser determinantes no uso e seleção destes serviços. Dentre os requisitos fundamentais de correção em ambientes de redes de longa distância (WANs), estão a confiabilidade, a disponibilidade e o desempenho.

As técnicas de replicação são normalmente empregadas para atender a estes requisitos, uma vez que o aumento de réplicas adiciona garantias de funcionamento correto do serviço, mesmo diante da ocorrência de um certo número de falhas, ataques e intrusões no serviço. As técnicas de replicação usadas na manutenção da disponibilidade dos serviços, com o acréscimo de outros mecanismos, fazem parte dos fundamentos da chamada Segurança de Funcionamento (*Dependability*) [Avizienis, *et al.*, 2004].

Este capítulo tem por objetivo apresentar alguns conceitos de Segurança de Funcionamento que são usados nos nossos trabalhos que envolvem sistemas distribuídos apresentados neste texto. Por outro lado, exploramos a literatura envolvendo a replicação Máquina de Estados (ME) [Schneider, 1990], envolvendo ambientes intrusivos e de larga escala. A nossa preocupação neste estudo é procurar soluções de algoritmos distribuídos que possam ser aplicadas na Internet a custos não proibitivos, mas mantendo as propriedades necessárias de serviços corretos.

### 2.1 CLASSIFICAÇÃO DE FALTAS

Assim como qualquer outro sistema computacional, um sistema distribuído também é passível de falhas parciais e deve conviver com as mesmas (falhas de serviços de seus componentes). Existem várias classificações de faltas considerando diferentes aspectos. Uma destas classificações, que é muito considerada em sistemas distribuídos, leva em consideração a **semântica de falhas**. Esta classificação é de grande importância para o projeto de algoritmos tolerantes a faltas. Ou seja, conseguir caracterizar o comportamento nas falhas de elementos faltosos é de grande utilidade no projeto de algoritmos e sistemas distribuídos. A partir destas informações, é possível quantificar e qualificar os mecanismos necessários para tolerar os tipos de faltas identificados e seus respectivos comportamentos.

Esta classificação, na separação dos tipos de faltas, faz uso das semânticas de falhas considerando os **domínios de tempo e de valor**. As faltas segundo suas semânticas de falha podem ser classificadas em cinco categorias diferentes [Cristian, 1991]:

- **Faltas de parada** (ou *crash*): são caracterizadas por falhas que provocam o travamento ou a interrupção total do componente faltoso, fazendo com que ele nunca consiga assumir um próximo estado (evolução bloqueada). Na ativação de uma falta de parada, o sistema (ou um subsistema) passa a não atender requisições de serviço e só atenderá novas requisições quando o mesmo for reiniciado e voltar a um estado normal (em que faltas não são ativas). Este tipo de falta envolve somente erros no domínio do tempo;
- **Faltas de omissão**: são aquelas que fazem com que um sistema não responda a algumas requisições de serviço. O seu serviço observado da interface apresenta um comportamento intermitente ou transitório, ou seja, atende algumas requisições e não responde a outras. As faltas de omissão são mais gerais que as de *crash* e também tem seus erros associados ao domínio de tempo;
- **Faltas por temporização**: determinam que um sistema responda a uma requisição de serviço fora de seu intervalo de tempo especificado para resposta válida. É importante ressaltar que falha por temporização só se caracteriza se a especificação do sistema impuser restrições temporais ao serviço. Estas faltas envolvem o domínio de tempo e também são chamadas na literatura de faltas de desempenho;
- **Faltas de valor**: são aquelas que ocorrem quando uma resposta é devolvida com um valor fora daqueles determinados pela sua especificação. As especificações de tempo são obedecidas (o tempo de resposta está dentro do intervalo de tempo especificado), logo, envolvem somente o domínio de valor. Estas faltas são por muitos autores incluídos dentro das faltas arbitrárias;
- **Faltas arbitrárias**: também chamadas de faltas **bizantinas** ou **maliciosas**. As falhas resultantes deste tipo de faltas envolvem os dois domínios citados (o de valor e o de tempo). Uma resposta a uma requisição pode estar correta segundo o domínio de valores, porém numa falha maliciosa pode provocar o envio da mesma fora do prazo especificado (também falta de temporização). Outra forma de ação maliciosa é devolver dentro do prazo um valor inadequado fora da especificação de valores (também falta de valor). O comportamento malicioso é difícil de ser detectado porque envolve o conhecimento



da aplicação e de sua semântica. Esta categoria engloba todas as classes de falhas citadas acima.

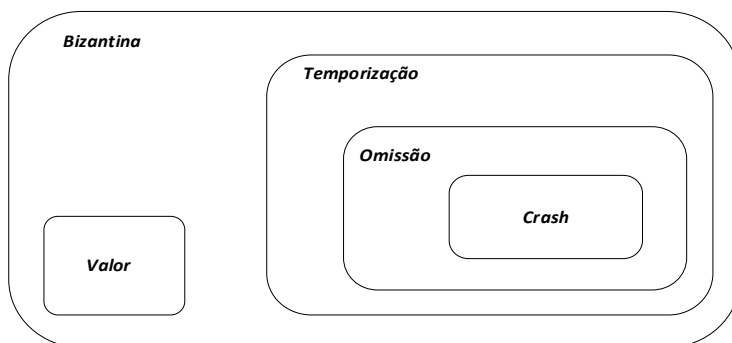


Figura 2.1 - Classificação da semântica de falhas.

A Figura 2.1 ilustra a classificação das falhas citada nesta seção, apresentando a relação entre os diferentes tipos de falhas. A falta por *crash* é a mais restritiva das classes enquanto a bizantina representa a menos restritiva, envolvendo os domínios de tempo e de valor. Esta última, devido a sua abrangência, demanda um custo alto em termos de recursos na obtenção de serviços tolerantes a falhas (bizantinas).

## 2.2 TOLERÂNCIA A INTRUSÕES EM SISTEMAS DISTRIBUÍDOS

O conceito de Tolerância a Intrusão foi introduzido em [Fraga e Powell, 1985] e surgiu inicialmente na comunidade de tolerância a falhas, mas vem se difundido no âmbito da segurança computacional [Correia, 2005]. O objetivo da tolerância a intrusões é garantir que um sistema continue o seu funcionamento correto mesmo que algumas partes desse sistema estejam comprometidas por ações maliciosas. É bom que se entenda que os conceitos de intrusão e de falhas bizantinas se confundem quando consideramos o contexto de segurança computacional (*security*). Invadir um sistema (intrusão) e corromper o mesmo (introduzir uma falta bizantina), de modo que se tenha um comportamento malicioso deste, faz com que estes conceitos se sobreponham: tolerar intrusões corresponde a tolerar também falhas bizantinas.

A tolerância a intrusão assume que um sistema distribuído apresenta vulnerabilidades permanentes, seus componentes podem ser atacados e que ainda, estes ataques terão sucesso modificando o comportamento de alguns destes componentes. Mesmo assim, fazendo uso de redundância, é possível garantir que o sistema como um todo

permanença seguro e operacional, ou seja, que este não falhe [Veríssimo, *et al.*, 2003] [Correia, 2005].

Considerando a manifestação das faltas maliciosas em um sistema computacional, em [Veríssimo, *et al.*, 2003] é apresentada uma taxonomia chamada de *Attack-Vulnerability-Intrusion* (AVI), que descreve os conceitos de ataque, de vulnerabilidade e de intrusão. A relação estabelecida nesta taxonomia entre estes conceitos é que a intrusão é o resultado de um ataque que obteve sucesso ao explorar uma vulnerabilidade encontrada no sistema. É importante ressaltar que o termo tolerância a intrusões passa a ideia de que uma intrusão é um tipo de falta que visa a corrupção do comportamento de um sistema ou componente. A Figura 2.2 ilustra as relações descritas entre os conceitos discutidos nesta seção.



Figura 2.2 – Relação entre ataque, vulnerabilidade e intrusão.

Um *ataque* é uma atividade movida por um indivíduo que tenta violar deliberadamente uma ou mais propriedades de segurança (confidencialidade, integridade e disponibilidade), explorando para isto as vulnerabilidades do sistema. Uma vulnerabilidade é uma oportunidade para o atacante; pode ser considerada como uma falta de projeto (*design fault*) que, ao ser explorada, pode provocar uma intrusão no sistema. A intrusão como resultado de um ataque bem sucedido do ponto de vista da confiabilidade do sistema pode ser encarada como uma falta em um componente de um sistema distribuído. Após uma intrusão, pode ocorrer um erro no estado do sistema que, quando não tratado, através de técnica de mascaramento, pode se propagar, resultando em falha do sistema como um todo. Logo, a aplicação de técnicas de redundância pode tornar o sistema tolerante a intrusões, evitando com isto, a propagação dos erros e de falhas no sistema distribuído.

Em síntese, pode-se afirmar que tanto intrusões como faltas bizantinas se referem a componentes maliciosos que podem determinar a falha do serviço de um componente de um sistema distribuído. O sistema deve continuar correto, mesmo que uma parte de seus componentes apresente qualquer comportamento malicioso.

Como área de pesquisa, a tolerância a intrusões se tornou alvo de grande interesse nesta última década [Veríssimo, *et al.*, 2003], [Reiser e Kapitza, 2007], [Júnior, *et al.*, 2010], [Bessani, *et al.*, 2007], [Sousa, *et*

al., 2008] sendo objetivo principal de diversos trabalhos e de grandes projetos relacionados à Segurança de Funcionamento (*Dependability*) em sistemas distribuídos.

A seção seguinte descreve os principais problemas clássicos relacionados com a tolerância a faltas e intrusões em sistemas distribuídos.

## 2.3 PROBLEMAS CLÁSSICOS DE SISTEMAS DISTRIBUÍDOS

### 2.3.1 Consenso

O problema do *consenso* em sistemas distribuídos faz parte dos chamados *problemas de acordo* e é parte fundamental de várias aplicações distribuídas. O consenso consiste em fazer com que todos os processos corretos envolvidos em uma computação distribuída terminem por decidir o mesmo valor, desde que este tenha sido previamente proposto por algum dos processos envolvidos.

Um *problema de consenso* envolve um conjunto  $P = \{p_1, p_2, \dots, p_n\}$  com  $n$  processos, onde cada processo  $p_i$  propõe um valor  $v_i \in V$ , sendo que ao final do consenso todos os processos corretos devem decidir de forma irrevogável e unânime sobre um dos valores contidos no conjunto dos valores propostos  $V$ . Formalmente, este problema é definido em termos de duas primitivas conforme [Hadzilacos e Toueg, 1994]:

- *propose* ( $P, v_i$ ):  $v_i$  é o valor proposto por  $p_i$  ao conjunto de processos  $P$ ;
- *decide* ( $v_i$ ): *upcall* que devolve o valor  $v_i$  decidido no conjunto  $P$  como consenso do grupo.

As primitivas acima listadas devem satisfazer as seguintes propriedades [Chandra e Toueg, 1996]:

- **Acordo**: se um processo correto decide  $v_i$ , então todos os processos corretos terminam por decidir  $v_i$ ;
- **Validade**: se um processo decide  $v_i$ , então  $v_i$  foi proposto por algum processo correto;
- **Integridade**: nenhum processo decide duas vezes;
- **Terminação**: todo processo correto termina por decidir.

A propriedade de acordo garante que todos os processos corretos decidem o mesmo valor, enquanto a validade assegura a consistência do

valor decidido. Além disso, as propriedades de acordo e validade juntas definem os requisitos de correção (*safety*) no consenso. A propriedade de terminação, por sua vez, garante o progresso do algoritmo de consenso (*liveness*).

O consenso pode permitir que processos falhos também participem das decisões. Esta condição só vale para faltas do tipo *crash*. Para que isto seja possível é necessária uma extensão nas propriedades acima. Na verdade, a propriedade de acordo é redefinida como segue [Charron-Bost e Schiper, 2004] [Raynal e Singhal, 2001]:

- **Acordo Uniforme:** Dois processos (corretos ou não) não decidem por valores diferentes.

Apesar das propriedades apresentadas até o momento serem consideradas suficientes para a implementação de protocolos de consenso diante de faltas de *crash*, em ambientes bizantinos, estas propriedades e a forma de definição do problema de consenso precisam ser redefinidas diante do comportamento arbitrário dos processos faltosos. Deste modo, em [Lynch, 1996] são apresentadas modificações nas propriedades de modo a permitir soluções ao problema de consenso em ambiente de faltas bizantinas:

- **Acordo:** todos os processos corretos decidem pelo mesmo valor;
- **Validade:** se todos os processos corretos iniciam com o mesmo valor  $v \in V$ , então  $v$  é o único valor de decisão possível para um processo correto.

Outras versões de validade são apresentadas na literatura para permitir consenso multivalorado em ambiente bizantino. Em [Doudou e Schiper, 1997] [Doudou, *et al.*, 1999] é mostrado como resolver este problema, definindo o problema de consenso de vetores.

Existem vários trabalhos na literatura para resolver o problema do consenso, tais trabalhos contemplam tanto faltas por parada (*crash*) [Lamport, 2001], [Charron-Bost e Schiper, 2004] quanto faltas bizantinas [Lamport, *et al.*, 1982] [Castro e Liskov, 1999] [Martin e Alvisi, 2006].

### 2.3.2 Difusão Atômica

A **difusão atômica** ou **difusão com ordem total** [Hadzilacos e Toueg, 1994], consiste em garantir que todos os processos corretos, pertencentes a um grupo, entreguem todas as mensagens difundidas neste

grupo e na mesma ordem. Desta forma, todos os processos devem ter a mesma visão do sistema e podem agir de maneira consistente sem comunicação adicional [Birman, 1996]. A difusão atômica é definida sobre duas primitivas básicas [Hadzilacos e Toueg, 1994]:

- *A-broadcast* ( $G, m$ ): a mensagem  $m$  é difundida para todos os processos pertencentes ao grupo  $G$ ;
- *A-deliver* ( $m$ ): a mensagem  $m$  é liberada (entregue) para a aplicação.

Estas primitivas devem satisfazer as seguintes propriedades [Hadzilacos e Toueg, 1994]:

- **Validade:** se um processo correto difundir  $m$  em  $G$ , então algum processo correto pertencente a  $G$  terminará por liberar  $m$ ;
- **Acordo:** se um processo correto pertencente a  $G$  entrega uma mensagem  $m$ , então todos os processos corretos pertencentes a  $G$  acabarão por liberar  $m$ ;
- **Integridade:** para qualquer mensagem  $m$ , cada processo pertencente a  $G$  entrega  $m$  no máximo uma vez e somente se  $m$  foi previamente difundida em  $G$  pelo seu emissor;
- **Ordem Total:** Se dois processos corretos  $p$  e  $q$  entregam as mensagens  $m$  e  $m'$  endereçadas ao grupo  $G$ ,  $p$  entrega  $m$  antes de  $m'$ , se e somente se,  $q$  também entregar  $m$  antes de  $m'$ .

Uma aplicação importante dos protocolos de difusão atômica é a implementação de tolerância a faltas através da replicação Máquina de Estado (ou replicação ativa) [Schneider, 1990]. O uso da difusão atômica como suporte da replicação ativa é importante para manter a consistência de estado e o determinismo entre as réplicas.

Foi mostrado em [Chandra e Toueg, 1996] para o modelo de faltas por parada, e em [Correia, *et al.*, 2005] para modelo de faltas bizantinas, que a difusão atômica e o consenso são problemas equivalentes onde os processos estão sujeitos aos tipos de faltas citadas.

### 2.3.3 Replicação Máquina de Estado

A replicação Máquina de Estado (ME) é sempre uma solução possível na concretização de serviços tolerantes a faltas. Esta abordagem pode ser usada na implementação de serviços tolerantes a faltas por parada (*crash*) como também a faltas bizantinas.

Na seção anterior, citamos o determinismo de réplica que deve garantir a mesma evolução e consistência entre as réplicas corretas de uma ME. Ou seja, com a coordenação adequada das interações dos clientes com as réplicas servidoras, a mesma evolução deve ser experimentada nos estados das várias réplicas do modelo. Para evidenciar esta evolução única das réplicas, foi formalizado o modelo ME em [Schneider, 1990], onde uma *máquina de estados* é definida por *variáveis de estado* que especificam o estado atual da mesma, e por *comandos* que modificam esse estado. Os comandos têm de ser atômicos, ou seja, não podem interferir uns com os outros. Deste modo, todas as réplicas corretas, quando submetidas à mesma sequência de comandos, em qualquer etapa de suas execuções, seguem a mesma sequência de estados.

Para execução de uma replicação Máquina de Estado é necessário satisfazer quatro propriedades:

- **Estado Inicial:** todos os servidores começam no mesmo estado;
- **Acordo:** todos os servidores executam os mesmos comandos;
- **Ordem Total:** todos os servidores executam os comandos segundo uma mesma ordem total;
- **Determinismo:** o mesmo comando executado no mesmo estado inicial gera o mesmo estado final (ou estado de saída do comando).

A primeira propriedade é simples de garantir, precisando apenas iniciar todas as réplicas com o mesmo estado, ou seja, todas as variáveis que representam o estado devem iniciar com os mesmos valores nas diferentes réplicas. A segunda e a terceira propriedades são facilmente supridas pela utilização de um protocolo de difusão atômica que deve garantir que todas as réplicas corretas recebam as mesmas requisições de comando e na mesma ordem. A última propriedade determina que o mesmo comando executado, com o mesmo estado inicial, implica no mesmo resultado ocorrendo nas diferentes réplicas. Ou seja, o resultado produzido (a mudança de estado) por este comando deve ser o mesmo nas diferentes réplicas corretas do sistema.

Nos últimos tempos, a comunidade de tolerância a faltas tem concentrado esforços na pesquisa de modelos e protocolos mais robustos. Diversas soluções têm sido apresentadas na literatura tendo como foco a tolerância a faltas bizantinas [Castro e Liskov, 1999] [Kotla, *et al.*, 2007] [Veronese, *et al.*, 2008], visando tornar aplicações mais resistentes (*resilience*) inclusive a intrusões (ataques bem sucedidos ao explorar as vulnerabilidades do sistema).

### 2.3.4 Faltas Maliciosas Independentes: Diversidade de Projetos

Dada a complexidade de sistemas computacionais em geral, evitar a ocorrência de faltas não é uma tarefa simples, tais sistemas, geralmente, são formados por diversos componentes interligados e se comunicando para a realização de tarefas. A ocorrência de faltas nesses componentes ou nos *links* de comunicação entre os mesmos pode levar ao mau funcionamento do sistema como um todo.

A utilização de técnicas de redundância e controle criptográfico constituem os mecanismos normalmente utilizados para a obtenção de sistemas tolerantes a faltas bizantinas [Verissimo e Rodrigues, 2001]. Quando se fala de intrusões o problema de garantir *independência de faltas* não é simples. Tanto a replicação como os controles criptográficos devem se basear na hipótese de que é mais difícil corromper a totalidade dos servidores do que uns poucos. Se o custo (grau de dificuldade) para conseguir ter acesso for o mesmo em todos os componentes de um sistema distribuído, não haveria a necessidade do uso da replicação e controle criptográfico. Um atacante bem sucedido em uma máquina poderia ter acesso a todos os componentes (máquinas) deste sistema distribuído de uma só vez.

Uma solução para esta questão é o uso de técnicas de *diversidade de projeto* que garantam a independência entre vulnerabilidades (faltas), evitando a presença de vulnerabilidades comuns nas diversas réplicas de serviço [Deswarte, *et al.*, 1998]. Cada réplica executando componentes de *software* diferentes, mas com o mesmo objetivo, minimiza a ocorrência das mesmas vulnerabilidades no conjunto total de réplicas. Diversos níveis de diversidade são sugeridos em [Deswarte, *et al.*, 1998] [Castro e Liskov, 2002] [Obelheiro, *et al.*, 2006].

## 2.4 ESTADO DA ARTE

Nesta seção, inicialmente começamos com a apresentação do algoritmo *Paxos* [Lamport, 1998] [Lamport, 2001] que embora trate de faltas benignas (*crashes*) serviu de base para grande parte de propostas algorítmicas que tratam com malícia e intrusões (faltas bizantinas). O motivo da apresentação do *Paxos*, nesta seção, é também porque o mesmo é base de parte do trabalho apresentado neste texto.

Na sequência, são então apresentadas as principais propostas para tolerância a faltas bizantinas encontradas na literatura. Tendo em vista a existência de uma literatura bem densa e diversificada nesta área, nós nos concentramos nesta seção em apresentar somente os trabalhos cujo foco

é a viabilidade prática de implementação de sistemas relacionados com a proposta desta tese.

### 2.4.1 Paxos

O *Paxos* é um conhecido protocolo para tolerância a faltas que permite a troca de mensagens com um conjunto de servidores distribuídos, através de comunicações assíncronas. Como suporte de replicação Máquina de Estados, o *Paxos* proporciona o acordo das réplicas sobre a ordem total de requisições de clientes enviadas para execução nos servidores replicados. No esquema do *Paxos*, um líder é eleito para coordenar o protocolo de consenso. Se as falhas do líder tornam o mesmo inacessível, os outros servidores elegem um novo líder em uma mudança de visão, permitindo com isto o progresso do protocolo.

O algoritmo *Paxos* na versão original foi desenvolvido para falhas de *crash* dos participantes do consenso. O *Paxos* requer pelo menos  $2f + 1$  servidores para tolerar  $f$  servidores falhos. Sendo um modo de falha benigna, ou seja, os servidores não são bizantinos e com isto, apenas uma única resposta precisa ser entregue ao cliente.

Três tipos de papéis são considerados no *Paxos* para ser assumidos pelos participantes neste algoritmo [Lamport, 1998, Lamport, 2001]:

- *Proponentes* – são os participantes que propõem os valores para decisão;
- *Aceitantes* – são os que escolhem (ou decidem) um valor entre os propostos para representar o consenso;
- *Aprendizes* – são aqueles que consomem (ou “aprendem”) o valor decidido.

Os participantes em aplicações distribuídas, quando do uso do *Paxos*, podem estar diferentemente distribuídos entre estes papéis. Mas, na maior parte das implementações deste algoritmo, todos os processos (participantes) do sistema desempenham estes três papéis [Marandi, *et al.*, 2010].

Este algoritmo se desenrola em rodadas assíncronas (*rounds*) e cada rodada é associada a uma *view* da instância de protocolo sendo executada e, por consequência, a um dado valor inteiro que identifica esta visão da instância. Em cada rodada  $r$  (ou visão  $r$  da instância) um proponente  $p_r$  é assumido como líder da rodada através do uso de uma função ou método que associe unicamente seu identificador com o número da rodada. Um exemplo destas funções é [Kirsch e Amir, 2008]:  $My\_Server\_id = r \bmod N$ , onde  $N$  define o número de servidores na replicação Máquina de Estados.



Este proponente líder serve de coordenador da rodada e todas as trocas de mensagens na rodada passam pelo mesmo. A instalação de uma *view* vai corresponder à escolha deste líder, sendo negociada na primeira fase (ou passo) de uma rodada. Uma vez definida a visão por instalar, o líder da mesma terá a responsabilidade de escolher um valor e enviar uma proposta aos aceitantes, os quais tentam fazer deste valor a decisão do consenso através de trocas de mensagens que definem a segunda fase do algoritmo. A decisão pode envolver mais *rounds* onde cada um destes *rounds* adicionais definem novos líderes que tentam fazer o conjunto da ME chegar à decisão que os *rounds* anteriores não conseguiram. Por fim, quando estabelecida, a decisão de consenso é enviada aos aprendizes (consumidores da decisão).

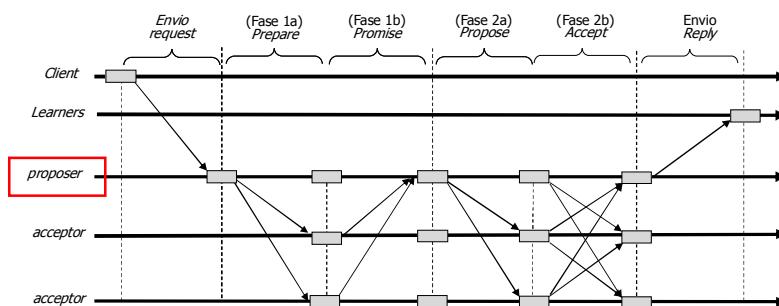


Figura 2.3 – Passo de uma rodada do protocolo Paxos.

A Figura 2.3 ilustra as trocas no algoritmo que devem garantir que um único valor proposto por um dos proponentes seja escolhido pelos aceitantes e aprendido por todos os aprendizes do sistema.

A escolha da *view* (ou líder da rodada) feita na fase 1 do protocolo é importante para que se chegue a decisão de consenso. Deve haver somente uma *view* instalada (somente um líder) na passagem para a fase 2, de forma que seja garantida que uma proposta será aceita e o sistema possa evoluir normalmente. Ou seja, podemos ter concorrência de *proposer's*, tentando instalar suas *views*, mas só um vai conseguir enviar proposta na segunda fase do protocolo.

A passagem de  $P$  réplicas como proponentes líderes, tal que  $P = f + 1$  garante que pelo menos um proponente correto<sup>1</sup> será (ou foi) líder em situação onde o número de líderes ultrapassa o limite  $f$  de réplicas faltosas.

<sup>1</sup> Réplica (ou processo) correta, neste caso, é aquela que durante toda a execução da instância de protocolo não sofre um *crash*.

Na segunda fase do protocolo, os aceitantes devem estabelecer o acordo sobre o valor proposto (proposta feita pelo *proposer* líder da rodada). Para isto, o número destas réplicas aceitantes não deve ser menor que  $2f+1$ .

Os proponentes devem manter suas informações em registros: a rodada com o maior identificador de rodada que coordenou, mantido em *crnd*, e o valor decidido em *round* especificado por *crnd* é registrado em *cval*. Um aceitante, por sua vez mantém, as suas informações nos seguintes registros: a maior rodada que o aceitante participou tem o seu número *round* em *rnd*, a maior rodada na qual o aceitante votou apresenta o seu número registrado em *vrnd* e, por sua vez, o valor que o aceitante votou na rodada indicada em *vrnd* é armazenado em *vval*.

Na Figura 2.3, uma rodada é mostrada com estas duas fases subdivididas:

- Fase 1: Um proponente, ao receber um valor  $v$  submetido por um cliente, sabedor que é líder da rodada  $i$  armazena este valor no seu registro de rodadas *crnd*. Este proponente como líder da rodada, tenta na sequência instalar a sua visão, fazendo com que uma maioria de aceitantes participe da rodada da qual pretende coordenar:

Fase 1a: O líder então inicia a rodada  $i$  ( $crnd \leftarrow i$ ) assumindo no começo  $cval \leftarrow \perp$  (seu registro de valores como indeterminado). O proponente líder, para obter esta maioria de réplicas participando da rodada da qual se propõe coordenar, envia uma mensagem de *prepare*, para cada um dos aceitantes, com os valores de seus registros *crnd* e *cval*.

Fase 1b: Um aceitante ao receber uma mensagem de *prepare* convidando para participar da rodada  $i$ , compara o valor de *crnd* enviado pelo líder com o seu registro *rnd*. Se  $crnd > rnd$ , o aceitante assume o  $i$  ( $rnd \leftarrow i$ ) como valor da rodada em que vai se engajar. Na sequência, compõe uma mensagem *promise* com o número de rodada  $i$  e os valores de seus registros *vrnd* (número da última rodada em que votou) e *vval* (último valor que votou) e envia esta mensagem ao coordenador. Um aceitante, uma vez que tenha respondido a uma mensagem de *prepare* do líder, não responderá a convites de outros proponentes a não ser que estes apresentem valores maiores de visão. Este é o caso em que *rnd* supera  $i$ , a mensagem *prepare* do proponente é então ignorada, significando que o *acceptor* já se engajou em outra proposta com número de rodada maior.

- Fase 2: Caso a maioria de aceitantes tenha se engajado na participação da rodada proposta, o líder escolhe uma proposta para enviar a todos os aceitantes.

Fase 2a: A escolha da proposta é feita baseando-se nas mensagens de confirmação de participação da rodada  $i$  (mensagens *promise*). Caso alguns *acceptors* tenham votado em rodadas anteriores (mandado valores de  $vval$ ), então o líder usa as mensagens recebidas de *promise* para escolher um valor para registrar em seu  $cval$ . Se mais de um *acceptor* envia valores, o líder pega como valor para o seu  $cval$  aquele que tiver maior  $vrnd$  (maior número de rodada em que o aceitante votou). Este mecanismo garante que somente um valor pode ser escolhido em uma instância de protocolo. Se nenhum aceitante apresentar valor que tenha votado em rodada anterior, então o coordenador assume como valor de  $cval$  o valor  $v$  enviado pelo cliente. Uma vez definido  $cval$ , o líder envia a mensagem *propose* para os aceitantes com  $crnd$  e  $cval$ .

Fase 2b: Um aceitante com  $rnd = i$ , ao receber mensagem de *propose* com um valor  $v$  na rodada  $i$ , assume  $vrnd = i$  e vota aceitando  $v$  ( $vval = v$ ). O aceitante envia então mensagens de *accept* confirmando ao líder e aos demais aceitantes que aceitou a proposta<sup>2</sup>. Caso as informações da mensagem de *propose* não sejam adequadas ( $i < rnd$  ou  $vrnd \neq i$ ), o aceitante ignora esta mensagem do líder.

Na sequência, depois da troca de mensagens *accept*, um *reply* é enviado ao “aprendiz” que corresponde ao originador da requisição ordenada e executada pela ME do esquema do *Paxos*, no caso, o cliente. Este *reply* é uma mensagem com o resultado da execução pela ME da requisição do cliente. A réplica servidora que recebeu originalmente a requisição do cliente é a responsável por este envio do *reply*. O cliente possui um mecanismo de temporização (de *timeout*) para tratar exceções relacionadas com a não recepção de mensagens de *reply*.

---

<sup>2</sup> O protocolo apresenta variações em relação a fase 2b. A primeira, original, todos os *servers* aceitantes enviam mensagens de *accept* somente ao líder da rodada. Depois este, para a evolução da Máquina de Estados precisa enviar um *confirm* a todos aceitantes para que estes executem a requisição do cliente. Neste caso, teríamos a latência de mais uma fase. Mas o custo assintótico do protocolo ficaria em  $O(n)$ . A outra versão é apresentada na Figura 2.3, onde os aceitantes respondem ao líder e aos demais aceitantes com suas mensagens de *accept*. Neste caso, diminuimos a latência de uma fase, porém o custo passa a ser quadrático ( $O(n^2)$ ). Neste último caso, embora mais cara, é mais robusta que a primeira: se o líder falha depois do envio do *propose*, a ordenação é assumida por todos os aceitantes corretos. No caso anterior, como o líder concentra os *accepts*, a decisão sobre a requisição do cliente deve ter que esperar um novo líder correto.

Diversas variações e otimizações para o algoritmo original explanado acima são encontrados na literatura [Hadzilacos e Toueg, 1994] [Lamport, 2006] [Lamport e Massa, 2004] [Gafni e Lamport, 2003].

### Trocas de Visão

Os *crashes* de aceitantes não provocam mudanças e o protocolo consegue evoluir normalmente desde que obedecidos os limites de falhas do sistema ( $|acceptors| \geq 2f+1$ ). O problema está nas falhas de *crash* de líderes. Cada falha de líder envolve troca de visão com a definição de novo líder. E o problema pode ser mais sério, com vários proponentes se acreditando líderes e concorrendo na primeira fase do protocolo, tentando instalar a sua visão. A fase de *prepare* resolve esta concorrência entre proponentes.

O protocolo envolve dois mecanismos de *timeout*. O primeiro com os aceitantes definindo um prazo para receberem a mensagem de *propose* do líder. Este prazo serve para a detecção de falhas do líder. A réplica que tem este prazo expirado envia mensagens de *view\_change* aos demais participantes, aumentando seu número de visão (número de rodada). Quando uma réplica servidora recebe uma *view\_change* que propõe uma visão maior, esta aumenta seu número de visão e envia também mensagem de *view\_change* com a visão mais alta do que aquela do líder falho. Na recepção de  $\lceil n/2 \rceil$  mensagens de *view\_change* de outras réplicas aceitantes, é caracterizada a condição que dá a partida para outra fase *prepare*, visando instalar uma nova visão que complete a execução da instância de protocolo. Quando a nova visão está instalada, o servidor volta a ativar o mecanismo de *timeout*, esperando receber a mensagem de *propose* do novo líder.

O segundo mecanismo de *timeout* envolve a recepção de mensagens *accepts* no número limite para a evolução da instância de protocolo (cada aceitante correto deve receber neste prazo no mínimo o limite de  $f+1$  mensagens de *accepts*). O tratamento desta exceção pode ser o reenvio da mensagem *propose* de modo a provocar os *accepts* com o *timeout* ajustado com valor de prazo maior. Ou na situação de bloqueio tentar provocar troca de visão.

### Propriedades do Protocolo

A propriedade de *safety* é sempre mantida pelo protocolo, mas a vivacidade só é satisfeita em *rounds* favoráveis. Um *round* é considerado favorável quando seu líder é correto e o sistema está num período de

sincronismo: as comunicações e computações ocorrem dentro de um período de tempo limitado. Nesta situação, um valor proposto pode ser aprendido dentro do período de um *round*. Caso um *round*  $r$  não seja favorável, um novo *round* é iniciado com um novo líder e assim sucessivamente até que um valor seja aprendido.

#### 2.4.2 PBFT: Practical Byzantine Fault Tolerance

O protocolo introduzido por Castro e Liskov [2002] conhecido como PBFT (*Practical Byzantine Fault Tolerance*), tem por finalidade prover um serviço tolerante a faltas bizantinas a partir da abstração de replicação de Máquina de Estado [Schneider, 1990]. Portanto, um conjunto de máquinas executa um serviço replicado determinista, juntamente com um protocolo de consenso tornando o serviço visto pelo cliente como robusto e seguro. Por ser uma proposta pioneira com o foco na viabilidade prática, uma série de otimizações é introduzida para prover eficiência ao sistema. Para isso, o sistema funciona na forma mais custosa que é na presença de réplicas maliciosas (intrusões), mas mantendo sua correção.

O modelo de sistema considerado apresenta um conjunto fixo de  $n$  réplicas de um serviço, dentre as quais no máximo  $f$  são faltosas. O conjunto de processos deve ser formado por  $n \geq 3f+1$  processos que implementam as réplicas do serviço. Cada réplica do PBFT executa uma cópia do serviço (que mantém um estado) e suas operações. As réplicas se comunicam umas com as outras na execução do protocolo de acordo, de modo a garantir que todas executem a mesma sequência de operações. A ME (Máquina de Estado) implementada proporciona ao sistema o mascaramento das falhas maliciosas de réplicas. Além disso, as mensagens são assinadas usando *Message Authentication Codes* (MACs), ao invés de usar criptografia assimétrica, no sentido de melhorar o desempenho nas trocas do sistema.

O algoritmo PBFT baseado na abordagem de replicação ativa (modelo ME) faz uso de difusões de mensagens através de uma réplica primária (ou líder). Ou seja, no PBFT, a réplica primária coordena a liberação das mensagens com propriedades de *atomic broadcast*.

Conforme ilustrado Figura 2.4, o funcionamento do algoritmo em modo normal (execução sem falhas) apresenta três fases: *pre-prepare*, *prepare* e *commit*. O objetivo das fases *pre-prepare* e *prepare* é propor uma operação e a ordem correspondente no conjunto de operações a serem executadas numa mesma visão  $v$  de participantes do serviço. A terceira fase envolve a decisão das réplicas na ordenação da requisição.

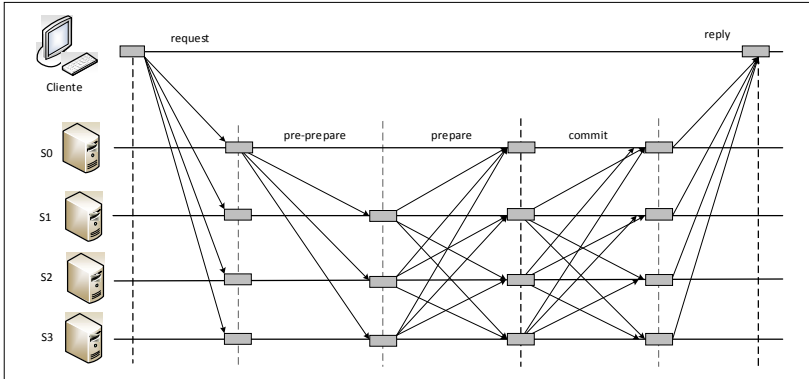


Figura 2.4 – Algoritmo PBFT em funcionamento normal.

Na fase *pre-prepare* o primário difunde, para todas as réplicas da visão (*membership* da replicação), o pedido recebido do cliente, com o número de sequência  $n$  que está propondo e o número  $v$  da visão atual do conjunto de réplicas. Uma réplica  $i$  aceita a mensagem se estiver na visão  $v$  e não tiver aceitado outra mensagem *pre-prepare* com os mesmos  $v$  e  $n$ . Caso aceite a mensagem e já tenha recebido o pedido do cliente correspondente, a réplica entra na fase *prepare*. Aceitar uma mensagem *pre-prepare* significa que a réplica está de acordo com o número de sequência atribuído pelo primário. Como consequência da aceitação, cada réplica difunde às demais uma mensagem *prepare* com um resumo criptográfico da mensagem e o número de sequência proposto pelo primário. Estas mensagens *prepare* são enviadas a todas as réplicas do sistema<sup>3</sup>. Quando uma réplica receber  $2f$  mensagens *prepare*, nas mesmas condições de correção e integridade das outras réplicas, terá assegurado que todas as réplicas corretas concordaram com o número de sequência  $n$  na visão  $v$  para a requisição do cliente. Então o pedido é dito como *preparado* para executar na visão  $v$ .

No entanto, para garantir a ordenação das mensagens mesmo durante uma troca de visão, a terceira fase de *commit* é necessária. Nesta fase, cada réplica  $i$  realiza a difusão da mensagem de *commit* para as outras réplicas quando tem um pedido já preparado. Quando uma réplica recebe  $2f + 1$  mensagens de *commit* de réplicas distintas, esta pode

<sup>3</sup> O modelo do PBFT assume *fair links*: ou seja, mensagens podem ser perdidas nestes canais. Se o emissor envia (e retransmite) mensagens um número infinito de vezes e o receptor neste canal recebe um número infinito de mensagens deste emissor.

executar a mensagem enviada pelo cliente, tão logo as mensagens anteriores (mensagens com número de sequência inferior ao da mensagem atual) tenham sido processadas. Concluída a execução, as réplicas enviam uma mensagem de *reply* diretamente ao cliente que solicitou.

Devido à premissa de sincronismo assumida – sincronismo terminal (*eventually synchronous*) – no sistema, um protocolo de troca de visão é incluído para garantir a propriedade de *liveness* que permite que o sistema evolua mesmo em caso de falha na réplica primária. Para isso, o algoritmo de troca de visão admite dois conjuntos de mensagens  $P$  e  $Q$  que contém as mensagens de *pre-prepare* e *prepare* de visões anteriores, respectivamente.

A troca de visão ocorre quando uma das réplicas secundárias suspeita que a réplica primária na visão  $v$  é faltosa. Então, a réplica passa para uma nova visão  $v + 1$  e envia uma mensagem *view\_change* para todas as réplicas informando sobre a sua constatação, juntamente com os conjuntos  $P$  e  $Q$ . Após esta mensagem ser emitida, a réplica deixa de participar da visão  $v$ . Com isso, ignora quaisquer mensagens, apenas aceitando as mensagens de *checkpoint*, *view\_change* e *new\_change*.

As réplicas aceitam mensagem *view\_change* apenas se possuírem visões menores que  $v + 1$ . As réplicas esperam por  $\lceil n/2 \rceil$  mensagens de *view\_change* para então enviarem uma mensagem de *view\_change\_ack* ao primário do *round*  $v + 1$ , reconhecendo esta nova visão. O novo primário  $p$  da visão  $v + 1$  monta um certificado de troca de visão composto por mensagem de *view\_change* e mais  $2f + 1$  mensagens *view\_change\_ack* recebidas durante o processo de troca de visão. Tal certificado atesta que a maioria das réplicas corretas concordou com a troca de visão. Além disso, permite a criação de um *checkpoint* para que seja possível definir os números de sequência para todas as mensagens pendentes da visão anterior contidas nos conjuntos  $P$  e  $Q$ . Por último, o novo primário difunde uma mensagem *new\_change* ao grupo de réplica contendo o *checkpoint* válido e as mensagens pendentes ordenadas. As demais réplicas aceitam e passam a nova visão desde que tenham aceitado a mensagem *view\_change* correspondente.

O bom desempenho do PBFT deve-se não apenas ao uso de criptografia simétrica, mas também a diversas otimizações. Entre as otimizações propostas temos a ordenação de requisições em lotes, execução por tentativa, resumo criptográfico como resposta e limpeza dos registros (*garbage collection*); estas são algumas das técnicas introduzidas por este trabalho. Muitas destas técnicas citadas tornaram se

correntes em trabalhos subsequentes ao deste artigo [Yin, *et al.*, 2003] [Merideth, *et al.*, 2005] [Martin e Alvisi, 2006] [Kotla, *et al.*, 2007] [Veronese, *et al.*, 2009]. Em nossa proposta iremos utilizar algumas destas técnicas de modo a alcançar o melhor desempenho do sistema.

### 2.4.3 Zyzyyva: Speculative Byzantine Fault Tolerance

Zyzyyva [Kotla, *et al.*, 2007] é um protocolo para a implementação de sistemas replicados tolerantes a faltas bizantinas também baseado na replicação de Máquina de Estado. A proposta deste protocolo é explorar a execução especulativa nos servidores de modo a reduzir o custo e simplificar o desempenho do sistema (em relação ao PBFT). O protocolo Zyzyyva é executado por  $3f + 1$  réplicas, sendo que no máximo  $f$  são faltosas.

Diferente do PBFT onde um primário propõe um número de ordem de execução para as requisições recebidas, no Zyzyyva, o primário propõe uma ordem as réplicas, e a mesma executa e responde a requisição de cliente de forma especulativa. Ou seja, a evolução das réplicas inicialmente se dá sem o uso de um protocolo de acordo na definição da ordem final na qual são executadas as requisições. Tais réplicas adotam, de forma otimista, a ordem de processamento da requisição proposta pela réplica primária e respondem imediatamente ao cliente. Porém, devido ao comportamento otimista, é possível que temporariamente o estado das réplicas fique inconsistente, fazendo com que o cliente receba respostas diferentes. Deste modo, a tarefa de garantir que as requisições sejam executadas de forma consistente fica a cargo do cliente que deve detectar e informar as réplicas sobre qualquer inconsistência. Para que estas ações sejam possíveis, as réplicas, ao enviar a resposta anexam informações do histórico de suas execuções para o cliente, permitindo que o cliente determine se as respostas e o histórico são estáveis.

A execução do protocolo ocorre da seguinte forma: o cliente envia a requisição apenas para a réplica primária e está repassa a requisição para o restante das réplicas (réplicas secundárias). As réplicas secundárias, ao receberem uma requisição, executam a mesma sem o devido conhecimento do acordo ou recebimento de tal mensagem pelas demais réplicas. Após a execução é enviada uma resposta correspondente diretamente ao cliente. Dependendo das respostas recebidas pelo cliente é possível tomar caminhos diferentes.



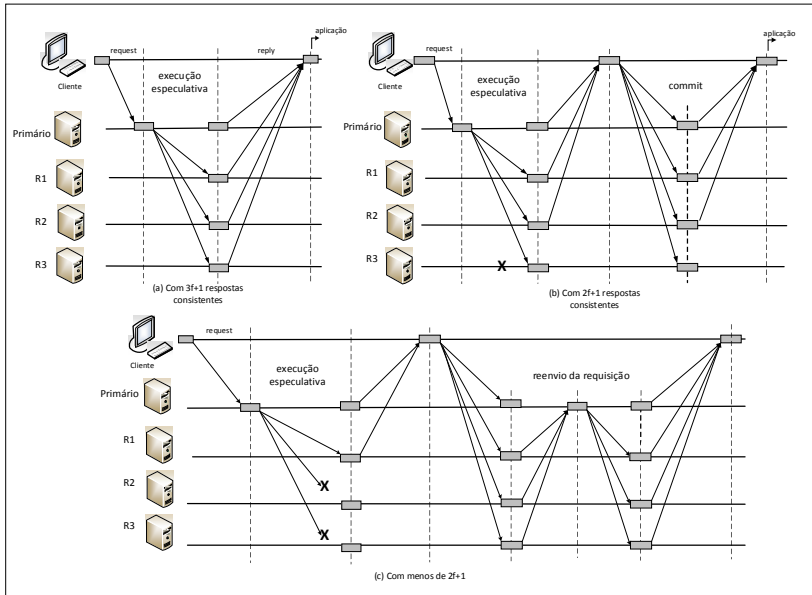


Figura 2.5 – Execução do protocolo de acordo do Zyzzyva.

No primeiro caso, o cliente recebe  $3f + 1$  respostas consistentes, ou seja, possuindo o mesmo resultado e histórico. Logo, o cliente considera que a requisição foi completada com sucesso, aceita como válida a resposta e entrega a mesma para a aplicação (Figura 2.5a). O segundo caso é quando o cliente recebe entre  $2f + 1$  e  $3f$  mensagens consistentes. Isto significa que o acordo foi estabelecido, mas existe divergência entre as réplicas. Logo, o cliente monta um certificado de *commit* composto por  $2f + 1$  respostas consistentes e envia às réplicas. Uma vez que  $2f + 1$  réplicas enviam uma resposta confirmando o recebimento do certificado, o cliente considera a requisição como completa e repassa o resultado para a aplicação (Figura 2.5b).

Outra situação que pode ocorrer durante a execução do protocolo é quando o cliente recebe menos de  $2f + 1$  respostas consistentes. Neste caso, o cliente reenvia a requisição para todas as réplicas, que redirecionam a mesma para o primário de modo a garantir que um número de sequência seja atribuído e a operação acabe sendo executada. (Figura 2.5c). Caso o cliente perceba qualquer inconsistência da réplica primária, como o número de sequência ou históricos diferentes, o cliente envia uma mensagem às réplicas junto com uma prova do mau comportamento da réplica primária, determinando uma possível troca de visão.

Além do protocolo de consenso, o sistema é composto também por um protocolo de *checkpoint* e um protocolo de troca de visão. O protocolo de *checkpoint* permite que as réplicas construam pontos de sincronização do serviço dentro de um intervalo de requisições processadas. Deste modo, permite que réplicas atrasadas sejam trazidas a um estado mais atual e também o descarte de mensagens e estados de mensagens antigas. *Checkpoints* auxiliam no protocolo de troca de visão.

As trocas de visões ocorrem quando o sistema está lento ou quando o primário é faltoso. Quando isso ocorre, a réplica que detecta um destes eventos, envia uma mensagem *i-hate-the-primary* para todas as réplicas. Ao receber  $f + 1$  votos de suspeita do primário, a réplica que propôs a troca, usa como prova estes votos e os envia junto a um certificado de troca de visão (mensagem *view-change*) para as demais réplicas. Uma réplica correta, ao receber um certificado de troca de visão válido, se junta à nova visão e confirma a troca. O protocolo de troca de visão é também responsável por coordenar a eleição de um novo primário.

#### 2.4.4 MinBFT: Minimal Byzantine Fault Tolerance

Em [Veronese, *et al.*, 2008], foi apresentado um protocolo tolerante a faltas bizantinas chamado *MinBFT*. A principal característica deste algoritmo é a redução nos números de passos de comunicação e de réplicas necessárias. Diferente do algoritmo do PBFT [Castro e Liskov, 2002] que necessita de  $3f + 1$  réplicas para tolerar  $f$  réplicas maliciosas, no *MinBFT*, são necessárias somente  $2f + 1$  réplicas para o acordo e a execução do serviço pelas réplicas do modelo ME. Além disso, o *MinBFT* apresenta outra vantagem sobre o PBFT que é no número de passos por *round*. O *MinBFT* reduz de 4 para 3 passos em comparação ao PBFT. O menor número de réplicas e de passos define uma menor ocupação de banda, reduzindo então a latência do algoritmo e o número de mensagens trocadas no acordo.

No modelo do *MinBFT*, uma das réplicas é designada como primária, ou seja, é a réplica que coordena a execução da requisição do cliente. As demais réplicas são identificadas como *backups*. Este papel de coordenação pode ser trocado em situações em que a execução da requisição não se resolva no *round* atual.

Além disso, no modelo proposto tanto as réplicas como clientes podem ser faltosos. Mas também são admitidos componentes confiáveis no sistema, ou seja, cada réplica de servidor contém um componente de hardware confiável (*TPM - Trusted Platform Module*) que fornece informações invioláveis através de um módulo USIG (*Unique Sequential Identifier Generator*). Cada módulo USIG contém um contador

monotônico e algumas funções criptográficas que são usadas para associar números de sequência (identificadores únicos) a requisições de cliente e as mensagens trocadas entre os servidores. O valor deste identificador que servirá de referência a estas informações, será único e seguirá uma sequência estabelecida.

Estes módulos USIG são então assumidos como invioláveis e confiáveis, sempre satisfazendo suas especificações, mesmo quando incluídos em servidores maliciosos (falsos). Deste modo, se um servidor malicioso decidir não enviar uma mensagem ou enviá-la corrompida, este jamais poderá enviar duas mensagens diferentes com o mesmo valor do contador USIG (ambas com o mesmo valor de identificador). Junto com este identificador é encaminhado um certificado gerado pelo USIG que impede a situação citada e serve para validar o identificador correspondente.

O modelo do *MinBFT* é considerado híbrido, onde convivem elementos corretos, maliciosos e confiáveis (os USIGs invioláveis). Como mostrado na Figura 2.6, o algoritmo *MinBFT*, em modo normal de operação (no lado do servidor), é executado em duas fases. Durante a primeira fase, chamada de etapa de *prepare*, a primária atribui um *número de sequência* assinado (identificador único - *UI*) para a requisição do cliente e envia a mesma para todas as réplicas em mensagens *prepare*. Este identificador é basicamente o valor do contador retornado pelo módulo USIG da réplica primária, acompanhado pelo certificado que garante a validade deste número de sequência.

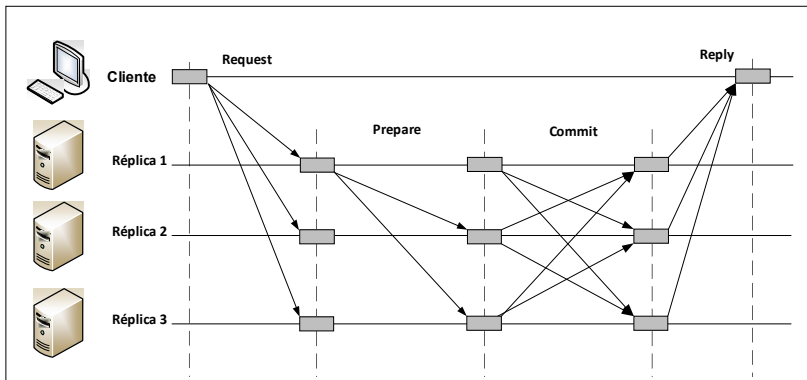


Figura 2.6 – Operação normal do algoritmo MinBFT.

Uma vez concluída a fase de *prepare*, resta apenas a confirmação pelas réplicas do servidor da aceitação da execução da requisição. Deste

modo, na última fase do algoritmo cada uma das réplicas difunde uma mensagem de *commit* às demais réplicas. A consolidação da ordem da requisição se dá na recepção de  $f + 1$  mensagens idênticas de *commit*, assegurando pela replicação a ordem estabelecida pelo contador da réplica primária para a requisição do cliente.

Cada mensagem *prepare* ou *commit* está associada ao identificador único obtido no USIG da primária. Quando a ordem de execução proposta pelo primário para a requisição é aceita, cada réplica envia diretamente uma mensagem de *reply* para o cliente contendo a resposta da requisição executada. O cliente, por sua vez, aceita o resultado caso receba  $f + 1$  mensagens *reply* iguais para a mesma requisição.

A ideia fundamental do *MinBFT* é que a réplica servidora que coordena o *round* use o seu módulo USIG para fornecer um identificador único assinado para a requisição do cliente, afastando qualquer possibilidade de que uma ordem única não venha a ser estabelecida, mesmo diante de comportamentos maliciosos. O certificado que acompanha o identificador da requisição atesta a validade do mesmo diante das réplicas que o recebem da primária. Não haverá no sistema outra requisição com o mesmo número de identificação. O objetivo é garantir que todas as réplicas não faltosas estejam de acordo sobre as mesmas requisições recebidas e que executem estas na mesma ordem imposta pelo USIG da réplica primária. A confiabilidade e a inviolabilidade dos módulos USIG garantem a correção do acordo e das ordenações destas requisições por parte das réplicas corretas.

O algoritmo *MinBFT* impõe algumas restrições quando a réplica primária é maliciosa, ou seja, não permite a esta réplica usar o mesmo número de sequência em duas requisições diferentes ou atribuir arbitrariamente um número de sequência muito alto de modo a levar as diferentes réplicas a estados inconsistentes. As propriedades do módulo USIG garantem estas restrições. No entanto, não impede que uma primária faltosa omita requisições recebidas do cliente ao não atribuir um número de sequência, através do USIG, para uma requisição. Entretanto, este comportamento malicioso da primária é detectado a partir das outras réplicas, determinando a execução de um protocolo de troca de visão e da escolha de uma nova primária. Ou seja, é estabelecido um novo *round* no algoritmo.

A solicitação da troca de visão ocorre quando uma réplica de *backup* suspeita, na visão  $v$ , do comportamento malicioso da réplica primária. As réplicas *backups*, após receberem uma requisição do cliente, estabelecem um *timeout* para receber da réplica primária o identificador e o certificado USIG, obtidos por esta última, a partir da requisição do

cliente. Se estas informações enviadas pela primária validarem a requisição correspondente, o *timeout* é cancelado. No caso dos *timeouts* expirarem, as réplicas *backups* passam a suspeitar da primária e, assim, iniciam uma troca de visão definindo um novo primário.

Quando uma das réplicas *backups* suspeita que o primário na visão  $v$  é faltoso, envia uma mensagem *req-view-change* para todas as réplicas informando sua decisão para a nova visão. Ao receber  $f + 1$  mensagens *req-view-change* as réplicas *backups* passam para a nova visão  $v + 1$  e difundem uma mensagem *view-change* contendo o último certificado de *checkpoint* estável. As réplicas corretas somente consideram mensagens *view-change* com  $f + 1$  certificados de *checkpoint*. A partir deste ponto, as réplicas param de aceitar mensagens na visão  $v$ .

Quando o novo primário recebe  $f + 1$  mensagens *view-change*, este usa as informações contidas nestas mensagens para definir a nova visão para o grupo. Com isso, o novo primário difunde uma mensagem *new-view* para todas as réplicas juntamente com um identificador USIG válido. As réplicas *backups*, para aceitar uma mensagem, verificam se o identificador foi gerado pelo módulo USIG do novo primário.

#### 2.4.5 Mencius: Efficient Replicated State Machine for WANs

Mencius foi proposto em [Mao, *et al.*, 2008] como um algoritmo de replicação ME tolerante a faltas por *crash* para ambiente de larga escala. O algoritmo proposto procura resolver o problema da sobrecarga da réplica líder que executa o protocolo de consenso. Os protocolos existentes, tal como o *Paxos* [Lamport, 2001] e o *Fast Paxos* [Lamport, 2006], apresentam limitações quando aplicados em ambiente de larga escala. Estes algoritmos normalmente contam com uma réplica primária que define a sequência de execução das requisições de clientes. No entanto, esta réplica líder se torna um limite ao desempenho do sistema, uma vez que a réplica primária tende a processar mais mensagens que as demais e ainda centraliza toda a coordenação do protocolo de acordo.

Neste contexto, em [Mao, *et al.*, 2008] é proposto um algoritmo de consenso tolerante a faltas, baseado no *Paxos*, para rede de larga escala. A ideia central do algoritmo é um esquema rotativo da réplica líder (multi-líder), ou seja, ao final de cada *round* do algoritmo um novo servidor é responsável por executar o algoritmo. O sistema é composto por  $n$  sítios interconectados por uma WAN. Cada sítio deve ser entendido como um domínio local formado, por exemplo, por uma rede local. Um sítio é formado por um servidor e um grupo de clientes que se comunicam através de uma rede local. Um cliente somente requisita uma operação ao seu servidor local e, em cada *round* de execução do algoritmo, é

designado um servidor diferente para ser a réplica primária. O algoritmo apresenta as mensagens, funções e etapas do *Paxos*: *proposers* (propõem os valores), *acceptors* (aceitam o valor a ser decidido) e *learners* (aprendem o valor escolhido).

O protocolo é executado da seguinte maneira: (1) Cada réplica (*proposers*) em seus sítios recebe requisições de seus clientes; (2) O líder atribui uma ordem de execução para a requisição recebida pelos clientes do seu domínio, através de uma mensagem *propose*; (3) Se não houver outros líderes ao mesmo tempo propondo requisições em nome de seus clientes na replicação, então os servidores reconhecem a requisição proposta pelo líder através de mensagens *accept*; (4) O líder recebendo mensagens de *accept* da maioria de servidores, difunde uma mensagem *learn* para informar aos outros servidores que a maioria das réplicas acordou sobre a ordem de execução. Após a confirmação da ordem de execução, com a mensagem de *learn*, os servidores executam a operação. O servidor primário irá enviar o resultado da requisição para o cliente (que pertence ao domínio do servidor).

O protocolo também apresenta outros comportamentos para os servidores. Se um servidor for o líder de um *round* e não tiver requisições de clientes de seu domínio para serem ordenadas. O servidor, neste caso, envia uma mensagem *skip*, informando que não possui mensagens para propor, dando a chance para outro servidor assumir a liderança. Outra mensagem foi definida no algoritmo para garantir o progresso do mesmo. Quando réplicas suspeitam da falha do líder, antes da definição de um novo líder, usam a mensagem *revoke* para revogar os direitos do líder faltoso na proposição de requisições de seu domínio.

#### 2.4.6 Steward: Byzantine Fault-Tolerance Replication to WANs

O Steward [Amir, *et al.*, 2006] é uma arquitetura hierárquica de ME que tem o objetivo de tolerância a falhas bizantinas em ambientes de larga escala. O Steward define dois níveis de protocolos: local e o global. No primeiro, é usado um algoritmo tolerante a falhas bizantinas baseado no PBFT [Castro e Liskov, 1999] que é executado em um sítio pelos seus servidores (domínio local). No segundo nível da hierarquia (nível mais alto), os sítios são reunidos em um acordo global, usando o algoritmo *Paxos* [Lamport, 2001]. Estes sítios estão distribuídos em uma WAN que com o algoritmo citado, constituem a nível global uma ME tolerante a falhas de *crash*. O número de sítios e réplicas necessários para garantir a execução do protocolo nos dois níveis hierárquicos são dados, respectivamente, por  $2g + 1$  sítios no nível global e para o nível local são necessárias  $3f + 1$  réplicas, onde  $g$  é o limite de falhas de sítios que o

sistema pode suportar e  $f$  é o número máximo de faltas toleradas pelos sítios de servidores<sup>4</sup>.

Para garantir a integridade e autenticidade das mensagens, o protocolo utiliza mecanismos de criptografia de limiar. Ou seja, para atestar que o conteúdo de uma mensagem foi originado em um determinado sítio, as mensagens enviadas a servidores e sítios são assinadas através de um mecanismo de assinatura de limiar  $(k, n)$ , onde  $k$  é o número mínimo de servidores necessário para gerar uma assinatura e  $n$  é o número de servidores onde *shadows* (ou partes) da chave privada compartilhada são distribuídos. Logo, o uso do protocolo PBFT e o esquema de criptografia de limiar permite confinar o comportamento malicioso localmente.

Um sítio é designado *site líder* em nível global, sendo este, portanto, encarregado de coordenar o protocolo *Paxos*, atribuindo um número de sequência para as requisições dos clientes. Em cada sítio há um servidor representante (*líder local*) que é responsável por coordenar o protocolo PBFT e o esquema de assinatura de limiar. Além disso, este líder local é o representante no nível global do seu domínio local.

Para garantir o acordo entres os sites tanto local como global são utilizados na arquitetura três protocolos: *Threshold-Sign*, *Assign-Sequence* e o *Assign-Global*. O primeiro é responsável em gerar a assinatura de limiar em uma requisição em nível local. O segundo protocolo é usado para gerar o número de sequência e criar uma *threshold-signed proposal* que é usado pelo protocolo *Assign-Global* para realizar o acordo global por todos os sítios no nível de hierarquia mais alto das MEs usadas. Os passos do protocolo são descritos na Tabela 2.1 abaixo.

O modelo do Steward provê então protocolos para tratar as falhas de réplicas que podem ocorrer tanto no nível global como dentro dos sítios de modo a preservar as propriedades de *safety* e *liveness*. Os dois níveis de protocolos envolvem trocas de visão em falhas de líderes e, para isto, são necessários em cada servidor dois temporizadores: *local\_t* e *global\_t*. Esses temporizadores são usados para detectar as falhas dos servidores nos níveis local e global, respectivamente. O disparo de qualquer destes

---

<sup>4</sup> O limite  $g$  representa o número de máximo de falhas que podem ocorrer em nível global. Corresponde a falhas de sítios (coordenadores de sítios deixam seus sítios indisponíveis a nível global por falha dos mesmos). O limite  $f$  por sua vez corresponde ao máximo de falhas de servidores em um sítio.

temporizadores pode implicar em trocas de visão em seus respectivos níveis de protocolo.

Tabela 2.1 – Passos do protocolo Steward.

<b>Protocolo para WANs</b>	
<b>Passos</b>	<b>Descrição dos Passos</b>
Passo 1	Um cliente envia uma requisição de serviço para o representante do sítio líder. Se o cliente não receber uma resposta no prazo, a requisição é então enviada a todos os servidores em seu sítio.
Passo 2	O representante do sítio líder recebe a requisição, invoca o protocolo <i>Assign-Sequence</i> para atribuir uma proposta de um número de seqüência global para a requisição, encapsulando a mesma numa mensagem <i>proposal</i> . O sítio então gera uma assinatura de limiar sobre a proposta através do protocolo <i>Threshold-Sign</i> , e o sítio líder envia o <i>proposal</i> assinado para os representantes ( <i>non-leader sites</i> ) de todos os outros sítios para a ordenação global.
Passo 3	Quando um sítio (o representante) recebe um <i>proposal</i> assinado, este encaminha a mensagem para o domínio local (via PBFT). Após receber a resposta local do PBFT, um servidor representante envia uma mensagem <i>Accept</i> para os representantes dos outros sítios. Esta mensagem <i>Accept</i> , antes do envio, é assinada através do protocolo <i>Threshold-Sign</i> que combina as assinaturas parciais no domínio local.
Passo 4	Quando o servidor representante recebe mensagens <i>Accept</i> dos outros sítios, este encaminha para os servidores locais.
Passo 5	Quando um servidor local correto recebe $2f+1$ mensagens <i>Accept</i> via PBFT, este aceita e executa a requisição, e envia a resposta ao cliente.

#### 2.4.7 EBAWA: Efficient Byzantine Agreement for WANs

O EBAWA proposto em [Veronese, *et al.*, 2010] é um algoritmo tolerante a faltas bizantinas em ambiente de larga escala. Neste algoritmo, a função de primário circula entre as réplicas. Ou seja, o primário muda depois que um lote de requisições pendentes é aceito para a execução. O algoritmo necessita de apenas dois passos de comunicação para alcançar o acordo e requer apenas  $2f + 1$  réplicas para tolerar  $f$  faltas maliciosas. Além disso, é introduzido um conjunto de mecanismos voltado para obter o melhor desempenho em ambiente de larga escala.

O algoritmo usa um componente confiável em cada servidor na replicação para fornecer informações invioláveis através de um módulo USIG (*Unique Sequential Identifier Generator*). O módulo USIG tem as mesmas características descritas na seção 2.4.4. Este módulo contém um contador monotônico e algumas funções criptográficas usadas para associar um número de seqüência (identificador único) a requisições que devem ser ordenadas e executadas pelas réplicas de uma ME.



O uso deste módulo confiável restringe o comportamento dos servidores faltosos, permitindo uma redução do número de réplicas de  $3f + 1$  para  $2f + 1$  em uma ME. Como mostrado na Figura 2.7, o algoritmo EBAWA, em modo normal de operação é executado em duas fases (*prepare* e *commit*). A execução do protocolo acontece da seguinte maneira: (1) O cliente envia uma requisição ao servidor mais próximo; (2) Este servidor atribui um número de sequência assinado (identificador único, UI) para a requisição do cliente e envia para todos os servidores numa mensagem *prepare*; (3) Quando recebe uma mensagem *prepare* válida do primário, o servidor envia uma mensagem *commit* para todas as outras réplicas; (4) Quando um servidor receber  $f + 1$  mensagens *commit* válidas, aceita a requisição, executa a operação e retorna uma resposta ao cliente; (5) O cliente aguarda por  $f + 1$  respostas iguais da requisição e completa a operação.

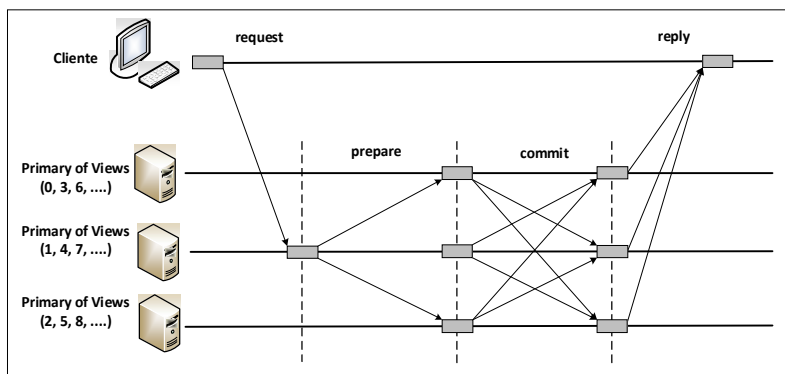


Figura 2.7 – Operação normal do algoritmo EBAWA.

Cada mensagem *prepare* ou *commit* está associada ao identificador único obtido do módulo USIG de cada servidor. Assim, é garantido que duas mensagens diferentes não tenham o mesmo identificador. Como mecanismo para melhorar o desempenho do algoritmo, o servidor primário ao receber as requisições de clientes, envia em uma única mensagem *prepare* um lote de requisições, no qual atribui um único UI para cada lote. Ou seja, um lote de requisições com um único UI é anexado a uma mensagem *prepare* que é enviada a outros servidores. Quando um servidor líder não tem uma requisição de cliente para ordenar e executar, o servidor, neste caso, envia também uma mensagem *skip* para as réplicas permitindo que outra réplica assuma a função de primário.

Mesmo usando o módulo USIG para limitar as operações de um primário faltoso, ainda assim uma réplica faltosa pode impedir o

progresso do algoritmo, ou seja, esta pode omitir o envio de mensagem *prepare* ou mensagem *skip* para as outras réplicas. Quando  $f + 1$  réplicas suspeitam que o primário é faltoso, uma operação *merge* é executada entre os servidores corretos para se chegar a um acordo sobre as requisições que já foram aceitas mas não executadas. Este acordo permite transferir estas requisições para uma próxima visão onde devem ser executadas.

#### 2.4.8 Replicação Máquina de Estados no ZZ

ZZ [Wood, *et al.*, 2011] é uma arquitetura tolerante a faltas bizantinas utilizando a virtualização para execução de replicação máquina de estado de servidores. A proposta apresentada aplica o conceito de separação do protocolo de acordo da execução, definido por Yin [Yin, *et al.*, 2003]. Dessa maneira, a arquitetura ZZ é composta por dois tipos de processos:  $3g + 1$  processos de acordo que definem uma ordem para as requisições recebidas dos clientes; e  $f + 1$  réplicas de execução que mantém o estado da aplicação e executam as requisições dos clientes. Estes processos de acordo e réplicas de serviço podem falhar independentemente, deste modo, são definidos limites separados para cada um desses tipos de processos (acordo e execução). Através da tecnologia de virtualização o ZZ executa os diversos processos em máquinas virtuais, no entanto, para manter o requisito de independência das falhas, em cada máquina física somente é executado um processo de acordo e uma réplica de execução.

A separação entre os papéis de acordo e réplicas de execução, como dito anteriormente sobre o método Yin, permite no ZZ a redução do número de réplicas de execução da ME para  $f + 1$  na tolerância a falhas bizantinas. A redução para  $f + 1$  réplicas de execução é possível em períodos livres de falhas. No entanto, quando ocorre um desacordo nas  $f + 1$  respostas destas réplicas de execução, temos uma indicação que existem réplicas maliciosas entre as mesmas. Nesse caso, ZZ inicia novas  $f$  máquinas virtuais ( $f$  réplicas que estão pausadas) para formar um novo quórum que agora assume uma ME com um total de  $2f + 1$  réplicas. Uma vez que, o quórum de  $f + 1$  réplicas que estavam em execução inicialmente, no máximo  $f$  destas podem ser incorretas. Para garantir uma maioria de respostas corretas ao cliente ( $f + 1$ ), precisamos somar estas novas  $f$  réplicas ativadas a partir do desacordo inicial.

Portanto, neste período de falha, o número de réplicas é incrementado para  $2f + 1$ , mas ao ser concluído o processamento das  $f$  novas réplicas e acordado o valor da resposta com  $f + 1$  respostas enviadas ao cliente, o número de réplicas deve voltar a mesma configuração inicial

de  $f + 1$  réplicas. As réplicas que apresentaram comportamento malicioso são então desligadas.

Na execução do protocolo, clientes enviam a requisição para o *cluster* de acordo (formado por  $3g + 1$  processos), com o objetivo de atribuir um número de sequência para essa requisição. Definida a ordem de execução para essa requisição, os servidores do *cluster* de acordo enviam essa para os servidores do *cluster* de execução, seguindo os mesmos princípios de [Yin, *et al.*, 2003].

As réplicas de execução executam uma requisição se receberem um conjunto de pelo menos  $2g + 1$  mensagens válidas enviadas do *cluster* de acordo. Essa execução irá gerar uma resposta, que será enviada para o cliente e também um relatório para o *cluster* de acordo. O relatório enviado para o *cluster* de acordo contém um resumo criptográfico da resposta assim como os objetos alterados durante a execução dessa requisição com seus resumos criptográficos. Os objetos alterados correspondem a quais alterações foram feitas no estado do sistema (*i.e.*, arquivos alterados em um sistema de arquivos, registros em um banco de dados).

Se o cliente não recebe as  $f + 1$  respostas esperadas (em *match*) das réplicas de execução, o mesmo reenvia a requisição para o *cluster* de acordo. Os processos deste *cluster*, se já tiverem recebido o relatório dessa requisição das réplicas de execução (se as réplicas já tiverem executado a requisição e enviado  $f + 1$  respostas em *match*), respondem para o cliente com este relatório de execução. Se o cliente receber  $g + 1$  resumos do *cluster* de acordo e tiver pelo menos uma resposta que corresponde a esse resumo criptográfico, este pode aceitar essa resposta como válida.

As réplicas armazenam em seus *logs* os relatórios de execução das requisições que são utilizadas para o envio no caso de retransmissões de clientes. Assim como em grande parte dos protocolos BFT, é necessário que as informações antigas sejam descartadas para evitar o problema do *buffer* infinito. Para que os servidores possam fazer a coleta de lixo de seus *logs* de mensagens, periodicamente é executada uma rotina de *checkpoint*. Além disso, a operação de *checkpoint* é também executada com o objetivo de salvar o estado mais atual da aplicação, o qual será obtido pelas novas réplicas que forem iniciadas em caso de falha.

Essa operação de *checkpoint* é feita pelos processos do *cluster* de execução. Um *checkpoint* é dito como estável se, uma réplica de execução recebe um certificado de *checkpoint*, contendo  $f + 1$  resumos criptográficos iguais para um mesmo *checkpoint*, vindos de diferentes réplicas. Recebendo um *checkpoint* estável, essa réplica pode descartar os *checkpoints* feitos anteriormente e efetuar a coleta de lixo de seus *logs*.

Além disso, este relatório de *checkpoint* estável é enviado para o *cluster* de acordo.

O *cluster* de acordo é responsável pela detecção de falhas em ZZ. No caso normal as réplicas de acordo aguardam por  $f + 1$  relatórios de execução correspondentes, vindos das réplicas de execução. Quando os processos do *cluster* de acordo não recebem essas respostas dentro de um tempo pré-determinado ou, recebem os  $f + 1$  relatórios, porém esses não são iguais, os processos do *cluster* de acordo enviam uma mensagem de recuperação para o *hypervisor* que controla as réplicas (máquinas virtuais) que estão pausadas. Quando o *hypervisor* de uma réplica que está pausada recebe  $g + 1$  mensagens pedindo ativação de novas réplicas, este então inicia as  $f$  novas réplicas.

#### 2.4.9 SMIT: Replicação ME usando Virtualização

SMIT [Júnior, *et al.*, 2010] é uma arquitetura para tolerância a faltas bizantinas que também usa máquinas virtuais. A arquitetura é composta por clientes e servidores que se comunicam através de rede local. Esses servidores por sua vez são executados em máquinas virtuais sobre uma única máquina física. Um dos diferenciais dessa arquitetura, além de executar os servidores em máquinas virtuais, é o fato de que a arquitetura utiliza uma área de memória compartilhada, denominada caixa postal (*postbox*), através da qual os servidores se comunicam, não necessitando da rede de comunicação para esse fim. Essa memória compartilhada é oferecida pelo sistema anfitrião, o qual é descrito como o sistema que fica sobre as máquinas virtuais, podendo ser representado pelo monitor de máquinas virtuais (VMM) seguindo os conceitos de virtualização. Outro diferencial dessa arquitetura é o número de servidores utilizados para prover o serviço, reduzindo o número de  $3f + 1$  da literatura original para  $2f + 1$  na arquitetura apresentada devido ao uso de conceito de componente confiável, que neste trabalho, é representado por uma memória compartilhada.

Para isso, ao invés de utilizar um protocolo de consenso com trocas de mensagem via rede, a arquitetura faz uso dessa área de memória compartilhada para prover a funcionalidade de ordenação das requisições, que serão executadas pelos servidores. Duas operações estão disponíveis para o acesso a essa área de memória. A operação *read*, que retorna um valor anteriormente escrito na memória compartilhada e a operação *append*, que insere um valor na memória compartilhada. Uma propriedade importante da memória compartilhada é que a mesma opera em modo *append-only*, ou seja, uma vez que um valor é inserido na memória o mesmo não pode ser alterado, evitando que uma réplica

maliciosa forneça valores diferentes para serem lidos pelas demais réplicas.

O algoritmo segue de forma que um desses servidores faça o papel de réplica primária, a qual vai definir uma ordem para a requisição, enquanto os outros façam o papel de réplica secundária, que são as réplicas que somente irão executar uma requisição ordenada. Os clientes enviam suas requisições para todas as réplicas. A réplica primária, por sua vez, escreve na memória compartilhada a sua proposta de ordem para a requisição recebida do cliente. As demais réplicas leem esse valor de proposição de ordem da memória e executam a requisição correspondente.

Uma vez que todas as réplicas leram a ordem da memória compartilhada e executaram a requisição, cada uma delas envia a resposta para o cliente. O cliente por sua vez aguarda que pelo menos  $f + 1$  mensagens com resposta válida iguais, vindas de réplicas diferentes para finalizar a operação.

## 2.5 CONSIDERAÇÕES

Apresentamos um resumo de alguns conceitos e terminologia da Segurança de Funcionamento considerando o contexto de sistemas distribuídos. Foram apresentados os principais problemas identificados na literatura de sistemas distribuídos relacionados a segurança de funcionamento. Estes problemas são importantes no desenvolvimento de qualquer aplicação distribuída que possua requisitos de confiabilidade e de segurança. Outro aspecto evidenciado foi de que protocolos desenvolvidos para solucionar estes problemas, dependendo do modelo de sistema considerado, podem apresentar complexidades diferentes, principalmente quando envolvem requisitos de tolerância a faltas e a intrusões.

Em relação aos protocolos de replicação Máquina de Estado com faltas bizantinas podemos observar que todos compartilham as mesmas características quando executados em modo de operação livre de faltas. E, certamente, o *Paxos* foi o grande modelo destes algoritmos.

O primeiro trabalho apresentado para tolerância a faltas bizantinas, o PBFT [Castro e Liskov, 1999] [Castro e Liskov, 2002], foi incluído por sua fundamental importância na área de tolerância a faltas bizantinas, pela sua abordagem com viabilidade prática, que são comprovadas através dos resultados, e as otimizações propostas que foram usados em outros trabalhos que se seguiram. Entre esses trabalhos temos [Yin, *et al.*, 2003], [Kotla, *et al.*, 2007], [Chun, *et al.*, 2007], [Correia, *et al.*, 2010], [Veronese, *et al.*, 2008] que apresentam soluções práticas com o objetivo

de buscar o menor custo e passos de comunicação em comparação à proposta apresentada no *PBFT*.

*Zyzyva* [Kotla, *et al.*, 2007] foi considerado por apresentar uma abordagem nova para o *PBFT* com o melhor desempenho considerando que todas as réplicas são corretas. Este protocolo usa a ideia de execução especulativa no contexto de faltas bizantinas com o intuito de tornar o sistema mais eficiente em execuções sem falhas. Entretanto, quando há presença de processos faltosos, sujeitos a perdas e atraso de mensagens, a participação do cliente passa a ser importante na execução do algoritmo. Há tem um aumento no número de fases, nas trocas de mensagens, de dois para cinco na obtenção do consenso sobre as requisições decididas para execução em presença de réplicas maliciosas.

Em [Veronese, *et al.*, 2008] foi apresentado o *MinBFT*, um protocolo de *BFT* mais eficiente em relação a protocolos mais tradicionais. Uma vez que, necessita de  $2f + 1$  réplicas para tolerar  $f$  réplicas faltosas e dois passos de comunicação para se chegar ao acordo entre as réplicas. A grande contribuição deste protocolo é o uso de um componente de hardware confiável, no qual fornece informações invioláveis através de um contador confiável (o módulo USIG). Além de fornecer números de sequência, o componente confiável gera certificados assinados que permitem atestar a validade destes números e controlar a atuação das réplicas primárias. Isto garante que réplicas não faltosas estejam de acordo sobre as mesmas requisições recebidas e que a execução ocorra na mesma ordem imposta pela réplica primária.

Em *Mencius* [Mao, *et al.*, 2008] foi apresentado um algoritmo para a replicação de estado da máquina em WANs para tolerar faltas por *crash*. Este algoritmo tenta resolver um problema da sobrecarga do servidor líder, executando um protocolo de consenso baseado em *Paxos*, usando um esquema rotativo da função de líder. No final de cada *round* do algoritmo um novo servidor/réplica assume a liderança no protocolo de acordo. Deste modo, o trabalho apresentado, tenta diminuir a sobrecarga no servidor líder, e a limitação da largura de banda disponível. Este protocolo implementa a ordem de requisições em um ME de sítios formados por um servidor e seus clientes.

Em [Amir, *et al.*, 2006] foi apresentado o protocolo *Steward* que tem como característica principal a proposição de uma arquitetura hierárquica através de dois níveis de algoritmos distribuídos para tolerar faltas bizantinas. Além disso, os servidores desta arquitetura estão distribuídos em sítios que desempenham o papel de um domínio local em uma WAN. Desta maneira, dentro de cada sítio (domínio local) um algoritmo tolerante a faltas bizantina baseando no *PBFT* é executada,

enquanto outro algoritmo tolerante a faltas baseado no *Paxos* é executado entre sites na WAN. Entretanto, esta solução do Steward apresenta algumas limitações que acarretam uma perda no desempenho. Ou seja, os protocolos usados mantêm uma réplica primária responsável pelas múltiplas trocas de mensagens entre as réplicas para acordar sobre a operação a ser executada, gerando uma sobrecarga do servidor na execução das operações.

Em [Veronese, *et al.*, 2010] foi apresentado o protocolo EBAWA que é caracterizado por um primário rotativo. O EBAWA é semelhante ao *Mencius* com a diferença que o primeiro implementa ME tolerante a faltas bizantina e possui alguns mecanismos (módulos USIGs) que permite um maior desempenho do mesmo.

A arquitetura ZZ [Wood, *et al.*, 2011] apresenta o efeito do uso da tecnologia da virtualização que torna a manipulação das réplicas mais dinâmicas. O fato da detecção das réplicas maliciosas e a respectiva remoção das mesmas faz com que o sistema permaneça sempre com réplicas de execução corretas. Apesar disso, os custos com as réplicas ainda continuam um pouco elevados, pois, como é definido um limite de no máximo uma réplica de acordo e uma de execução por servidor físico, caso se utilize um protocolo de acordo baseado no PBFT, são necessárias pelo menos  $3f+1$  servidores físicos.

O SMIT [Júnior, *et al.*, 2010] é uma arquitetura para tolerância a faltas bizantinas que usa máquinas virtuais. A arquitetura ME desta proposta possui seus servidores em máquinas virtuais. E a execução do protocolo de acordo entre servidores é via memória compartilhada, usando o conceito de *postbox*. Esta replicação ME sobre máquinas virtuais que executam sobre uma mesma máquina física. Esta proposta está também baseada no conceito de *elemento confiável* que faz com que a necessidade de  $3f+1$  réplicas para o acordo caia para o número de  $2f+1$  réplicas. Esta proposta não usa as facilidades de reconfiguração de máquinas virtuais para tratar com réplicas maliciosas e não tolera o crash do hospedeiro físico do esquema.

Na literatura também são apresentados outros trabalhos baseado em sistemas híbridos tais como [Correia, *et al.*, 2002], [Júnior, *et al.*, 2010], [Reiser e Kapitza, 2007] onde abordam modelos híbridos de falhas, tratando separadamente diferentes tipos de faltas. Para esta separação, estes sistemas apresentam, em cada réplica, componentes locais seguros. Estes componentes seguros são interconectados aos componentes equivalentes de outros membros via redes de comunicação próprias que permitem consensos menos complexos, considerando usualmente só faltas de crash nestas redes especiais. [Júnior, *et al.*, 2010]

e [Reiser e Kapitza, 2007] utilizam esta abordagem para tolerar faltas bizantinas fundamentadas na tecnologia de virtualização. Na Tabela 2.2 é mostrado um comparativo das principais diferenças dos algoritmos mencionados. De qualquer modo são propostas diferentes que apresentam características interessantes, mas também limitações.

Tabela 2.2 – Comparativo entre as diferentes propostas.

	PBFT	Zyzyva	MinBFT	Mencius	Steward	EBAWA	ZZ	SMIT
<b>Passos de Comunicação</b>	4	2	3	3	2/4	3	4/2	2
<b>Tipo de Ambiente</b>	LAN	LAN	LAN	WAN	WAN	WAN	LAN	LAN
<b>Tipo de Faltas</b>	Bizantino	Bizantino	Bizantino	<i>Crash</i>	Híbrido	Bizantino	Bizantino	Bizantino
<b>Número de Réplicas</b>	$3f+1$	$3f+1$	$2f+1$	$3f+1$	$2g+1/3f+1$	$2f+1$	$3g+1/f+1$	$2f+1$

## 2.6 CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou conceitos fundamentais necessários no desenvolvimento de aplicações distribuídas com requisitos de segurança e confiabilidade, bem como os problemas relacionados com a tolerância a faltas e intrusões em sistemas distribuídos.

Os problemas e algumas soluções colocadas aqui são importantes quando pensamos no contexto de nossos trabalhos que é aquele de aplicações e serviços em redes de longa distância. Os conceitos sobre Serviços Web que complementam a nossa experiência serão tratados no próximo capítulo.



### 3 CONCEITOS EM ARQUITETURAS ORIENTADAS A SERVIÇO

A Internet é atualmente uma via essencial para a integração de sistemas e aplicações. Mas a mesma não fornece, entre seu conjunto de protocolos, suporte, com padrões aceitos universalmente, para os níveis mais altos, próximos das aplicações. Isto permitiria, além da comunicação, a integração de sistemas computacionais e aplicações de forma simples.

A integração de aplicações e sistemas via rede mundial sempre foi complexa e onerosa. É para corrigir esse cenário que surgiram tecnologias como a de *Serviço Web* [Booth, *et al.*, 2004], cuja função principal é a interoperabilidade entre aplicações nestes ambientes de larga escala, fazendo uso de padrões consolidados. Esta interoperabilidade deve ser independente de plataformas, linguagens e máquinas, permitindo desta forma a integração de aplicações na rede mundial, sem as dificuldades do trato da heterogeneidade pelas mesmas.

Esta tecnologia integradora é baseada em conceitos da Arquitetura Orientada a Serviços (SOA), fornecendo meios para publicação, invocação e localização de serviços. Esta tecnologia e suas especificações têm criado muitas perspectivas para que aplicações distribuídas e suas qualidades de serviço em redes como a Internet.

Uma das principais características que tornam os serviços web uma tecnologia interessante é a facilidade de composição. Assim, problemas complexos podem ser resolvidos por aplicações criadas a partir da combinação, em uma ordem conveniente à solução do problema, de funcionalidades fornecidas por serviços básicos [Milanovic e Malek, 2004]. Essa abordagem facilita o desenvolvimento de aplicações e o reuso de serviços, diminuindo assim o custo e o tempo para a criação de novas soluções. Além disso, os serviços *web* envolvidos em uma composição podem ser providos por organizações distintas, permitindo um maior nível de integração entre parceiros de negócio, por exemplo.

Neste capítulo, será abordado de maneira resumida o modelo de desenvolvimento de *Web Services*. Inicialmente, são apresentados conceitos da Arquitetura Orientada a Serviços e os seus principais benefícios. Na sequência, são descritos os conceitos mais importantes que formam a noção de composição de serviços web. Como parte importante deste trabalho, literatura corrente sobre a orquestração descentralizada é também apresentada em detalhes.

### 3.1 ARQUITETURA ORIENTADA A SERVIÇOS

A Arquitetura Orientada a Serviços (*Service Oriented Architecture* – *SOA*) pode ser definida como uma caracterização de sistemas distribuídos, na forma de serviços. Estes serviços têm suas funcionalidades expostas via descrição de uma interface, permitindo a publicação, localização e a invocação dos mesmos por meio de um formato padronizado [Booth, *et al.*, 2004].

A *SOA* utiliza, portanto, o conceito de serviço como elemento fundamental na concepção de aplicações distribuídas. É uma arquitetura que introduz uma infraestrutura de serviços que permite aplicações (através de seus serviços) se comunicarem entre si. Estas comunicações podem envolver simplesmente a troca de dados ou ainda, a ativação de dois ou mais serviços integrados em alguma atividade específica [Zdun, *et al.*, 2006].

Esta arquitetura define três tipos de papéis: (1) Provedor de Serviço – entidade que cria o serviço e é responsável por publicar a interface do mesmo no registro de serviços e ainda atender as requisições originadas pelos clientes; (2) Serviço de Registro – repositório utilizado na publicação e localização de interfaces dos serviços; e (3) Solicitante do Serviço – parte da aplicação (que pode ser também um serviço) que envia requisições para execução de serviços aos seus correspondentes provedores. Cada participante da arquitetura pode ainda assumir um ou mais papéis, podendo ser, por exemplo, um provedor e ainda um cliente de outros serviços.

A *SOA* também descreve as interações entre esses papéis, através de operações como: publicar, localizar e invocar. A Figura 3.1 ilustra a colaboração entre os participantes da *SOA*, identificando os papéis e as operações executadas para o acesso a um serviço. No caso, o cliente efetua, através de suporte do diretório de registros, a busca por informações do serviço, especificando as características desejadas do mesmo. Quando o registro do serviço desejado é localizado, informações de interface e a localização deste serviço são retornadas ao cliente. Por fim, com o uso destas informações, o cliente faz a invocação ao serviço desejado.

As execuções dos serviços desejados determinam trocas de mensagens bem definidas entre cliente e o provedor de serviço. A existência de formatos padrões bem especificados de mensagens garante aos serviços a neutralidade da tecnologia e permite que provedores e clientes utilizem diferentes tecnologias nas implementações das camadas inferiores.

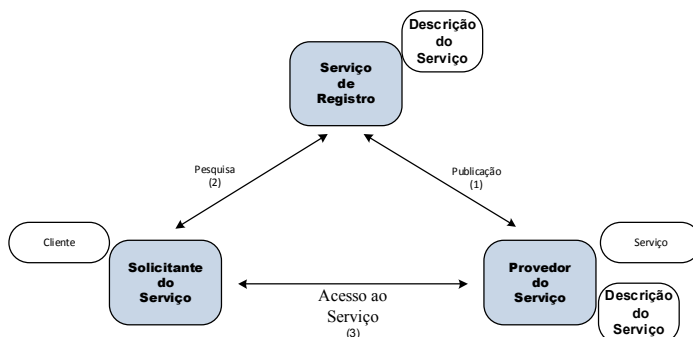


Figura 3.1 – Interação entre as entidades da SOA.

As interfaces de serviços são um contrato entre o provedor do serviço e o cliente, sendo que este contrato estabelece: um conjunto de operações públicas do serviço, o protocolo que deve ser usado na comunicação com o serviço, os parâmetros e valores de retorno de uma requisição e meios para tratar possíveis exceções.

### 3.2 SERVIÇOS WEB

A tecnologia de Serviços *Web* (*Web Service* – WS) é a representante mais notória das implementações da arquitetura SOA. Nesta arquitetura, serviços *web* são identificados através de um URI (*Uniform Resource Identifier*). Estes serviços devem ser independentes de linguagens, sistemas operacionais e arquiteturas de máquinas. A principal característica desta tecnologia é a de estar fundamentada sobre padrões abertos tais como o XML (*eXtensible Markup Language*) e o HTTP. Outra característica associada a esta tecnologia é a garantia da interoperabilidade entre clientes e provedores de serviços, sem a necessidade de um conhecimento prévio de plataformas de execução ou de implementações, necessitando apenas conhecer a interface do serviço *web* – sua especificação WSDL (*Web Services Description Language*). Além disso, esta tecnologia apresenta a capacidade de transpor os tradicionais filtros de pacotes, os *firewalls*, sempre que os seus protocolos são mapeados sobre o HTTP.

Existe alguma controvérsia na literatura sobre os conceitos de objeto e de serviço. Para [Vogels, 2003], a tecnologia de *Web Services* vem sendo usada em áreas onde soluções com objetos distribuídos falharam no passado. Esses dois conceitos possuem algumas características em comum, como uma linguagem de descrição para o serviço (IDL em CORBA [OMG, 2000] e WSDL para serviços *web*)

usam mecanismos semelhantes para registro e localização de objetos ou serviços. Por outro lado, em sistema de objetos distribuídos existe a ideia de se manter o estado da informação durante todo o ciclo de vida do objeto, permitindo uma computação distribuída *stateful*. Enquanto, em serviços *web* não é oferecida qualquer informação em relação ao estado (*stateless*) durante o ciclo de vida de uma invocação. Esta característica da última tecnologia se deve ao fraco acoplamento do mesmo. Clientes não interagem por meio de sessões de comunicação com serviços.

Ambientes como uma rede local são caracterizados muitas vezes pela homogeneidade de plataforma e por possuírem tempos de latência pequenos. Neste caso, a tecnologia de objetos distribuídos se apresenta como a mais apropriada para este tipo de ambiente. Já os serviços *web* são mais adequados em ambientes como a Internet, caracterizados pela impossibilidade de manter forte acoplamento por longos períodos. Outros aspectos também contribuem para estas escolhas. A tecnologia de serviço *web* oferece um suporte bastante consistente para a integração de ambientes de aplicação heterogêneos, enfatizando muito as relações de negócios entre empresas (*Business to Business – B2B*). Com o uso desta tecnologia e suas especificações é possível que estas relações entre empresas sejam estabelecidas de maneira simples e dinâmica.

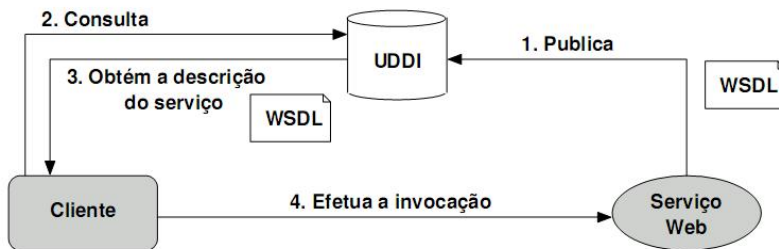


Figura 3.2 – Colaboração típica na arquitetura dos serviços *web*.

Um serviço *web* tem suas requisições e respostas trafegando sobre um protocolo de transporte que usualmente é o HTTP. É através deste protocolo que mensagens de aplicação, codificadas em texto XML, devem ser enviadas devidamente encapsuladas através do protocolo SOAP [Gudgin, *et al.*, 2003]. O protocolo SOAP implementa as interações *request/reply* (e outras mais elaboradas) de um modelo cliente/servidor. Entre outras características agregadas a essa arquitetura, destacam-se a linguagem de descrição de serviços *web* – WSDL

[Chinnici, *et al.*, 2007] e o repositório no qual podem ser publicados e localizados todos os serviços – UDDI [Clement, *et al.*, 2004].

A Figura 3.2 mostra uma versão da Figura 3.1 evidenciando alguns destes componentes desta tecnologia. Inicialmente (passo 1), um provedor de serviço publica a interface WSDL no UDDI (o repositório de registros de serviços nesta tecnologia), tornando assim o serviço visível para os possíveis clientes. No passo 2, o cliente realiza uma pesquisa por um serviço que corresponda às suas necessidades e assim, no passo 3, recebe a interface WSDL do serviço que possui as características desejadas. E por último, no passo 4, o cliente invoca o serviço desejado, respeitando os formatos de mensagens descritas pela interface obtida anteriormente, sendo que esta invocação é feita através da troca de mensagens SOAP.

### 3.3 COMPOSIÇÃO DE SERVIÇOS

Uma composição permite a construção de novos serviços usando aqueles já estabelecidos. Para o cliente, a noção de serviço é a mesma, sendo este composto ou não. Na literatura existem duas abordagens para a composição de serviços: orquestração (*orchestration*) e coreografia (*choreography*) [Peltz, 2003]. A primeira abordagem, a orquestração, descreve as interações entre serviços desta composição, através das trocas de mensagens, incluindo a ordem em que estas devem ocorrer. A imposição desta ordem de interações é concretizada por um coordenador central que é um dos participantes da composição. Portanto, neste caso, é bom que se explicita a existência de uma figura central controlando os demais participantes do chamado processo de negócio.

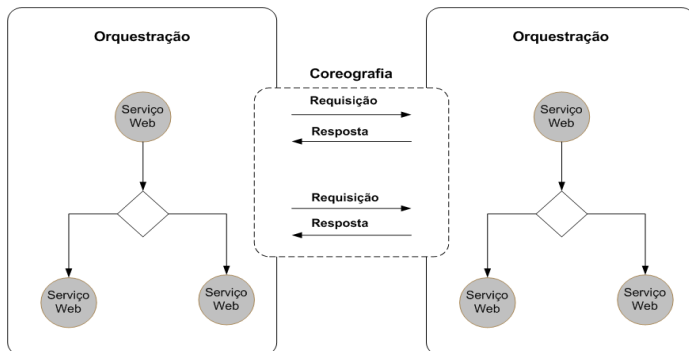


Figura 3.3 – Orquestração e Coreografia.

A coreografia, que é a outra forma de composição, descreve uma seqüência de troca de mensagens segundo uma perspectiva global. Em uma coreografia cada serviço *web* envolvido desempenha seu próprio papel, não existindo, portanto, a figura de um coordenador centralizado.

Diversas propostas de padrões para orquestração e coreografia de serviços *web* vêm sendo desenvolvidas. Dentre estas, destacam-se WS-BPEL (*Web Service Business Process Execution Language*) [Arkin, *et al.*, 2007] e WS-CDL (*Web Service Choreography Description Language*) [Ross-Talbot e Fletcher, 2006]. A Figura 3.3 situa as duas abordagens para composição considerando suas características conceituais. Já a Figura 3.4 mostra o posicionamento das linguagens de composição de serviços (WS-BPEL e WS-CDL) em relação à estratificação de especificações que formam a infraestrutura de serviços *web*. A seção seguinte descreve os mecanismos de composição presentes na tecnologia de serviços *web*, segundo os conceitos de coreografia e orquestração.

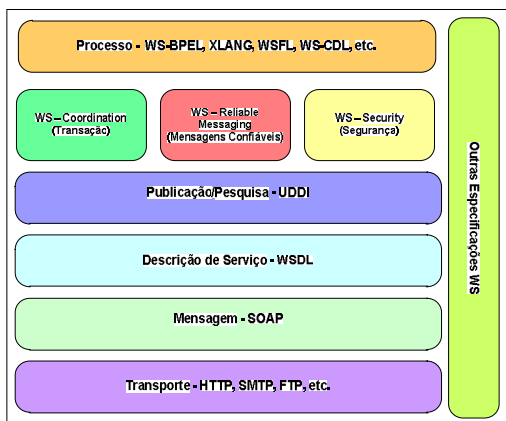


Figura 3.4 – Infraestrutura de serviços em *Web Services* [Tai, *et al.*, 2004].

### 3.3.1 Coreografia

A integração de sistemas requer mais do que a simples habilidade de conduzir interações usando protocolos padronizados de comunicação. É importante que em um processo de negócio, quando são envolvidos múltiplos parceiros seja definido um acordo sobre as interações, baseado em modelos padrões.

Coreografia é tipicamente associada à troca de mensagens públicas que ocorrem entre múltiplos serviços *web*, em vez de um processo de negócio específico coordenado por um único elemento centralizador. A coreografia, portanto, não contém o papel de um coordenador central.

Além disso, a coreografia define o tipo de política para as “regras contratuais” (por exemplo, protocolos de mensagens) necessárias nas relações implementadas entre parceiros em uma arquitetura de serviço *web*. Deste modo, as partes envolvidas devem possuir interfaces (contrato) especificando as trocas mútuas, o que corresponde a uma visão estática destas trocas entre serviços.

Em uma coreografia cada parte envolvida no processo de negócio define de forma colaborativa sua participação [Peltz, 2003]. A coreografia é descrita pelo comportamento visível dos serviços envolvidos, isto é, pelas trocas de mensagens de todos os envolvidos. Cada serviço *web* participante do processo de negócio sabe exatamente quando executar as suas operações e com quem interagir. A coreografia prioriza um esforço colaborativo concentrado na troca de mensagens em um processo de negócio. Todos os participantes da coreografia têm consciência do seu papel durante um processo de negócio. Com isso, a coreografia é utilizada para definir as regras de uma colaboração entre os participantes de um processo de negócio sem revelar detalhes internos dos participantes. As descrições das interfaces permitem que um participante especifique quando e quais informações serão enviadas para os seus parceiros [Mendling e Hafner, 2008].

A vantagem com o uso desta abordagem é permitir que diferentes equipes de programadores de uma organização desenvolvam diferentes serviços capazes de interoperar, deste que ambos sejam compatíveis com um mesmo contrato previamente estabelecido. Isso possibilita uma maior liberdade para implementar ou modificar um serviço sem afetar o protocolo (trocas de mensagens) definido na coreografia.

A WS-CDL (*Web Services Choreography Description Language*) [Kavantzias, *et al.*, 2005] é uma proposta para descrever coreografias. É uma linguagem baseada em XML e que tem por objetivo descrever as trocas públicas de mensagens que envolvem múltiplos participantes, onde mensagens trocadas ordenadamente resultam na execução de determinada atividade distribuída.

### 3.3.2 Orquestração

A composição de serviços permite especificar como um conjunto de serviços *web* deve ser usado para realizar funções mais complexas, tal como, um processo de negócio. É neste cenário que o conceito de orquestração se mostra bastante apropriado. Na orquestração, o controle sobre a composição evolui segundo a visão de apenas uma das partes envolvidas.

Neste contexto, a orquestração propõe um mecanismo de composição ou agregação de serviços visando processamentos distribuídos mais complexos. Composições obtidas segundo a abordagem de orquestração são importantes em arquiteturas orientadas a serviço, pois encapsulam logicamente diversos serviços de forma transparente, fazendo com que os mesmos pareçam um só para seus clientes. Outros aspectos podem ser associados a composições do tipo orquestração. Um exemplo sempre citado é o tempo de desenvolvimento de novos serviços que é fortemente reduzido em função do alto índice de reuso de código.

A orquestração, como já foi dito, possui um processo central (orquestrador) que mantém o controle sobre os serviços *web* da composição. Este orquestrador define a ordem das execuções das diversas operações entre os serviços que compõem a orquestração. Os serviços *web* participantes da composição não são cientes que fazem parte de uma composição em um processo de negócio e nem são criados para tal objetivo. Apenas o orquestrador, que é gerado para esta finalidade, sabe da composição de serviços e do plano de execuções na mesma.

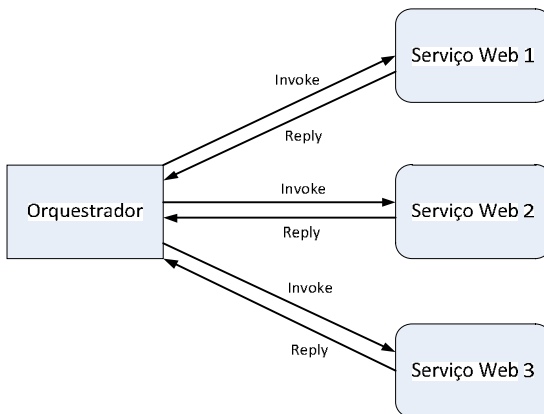


Figura 3.5 – Exemplo de orquestração.

A orquestração de serviços define então um fluxo de execução (*workflow*) de um processo de negócios a partir de um *script* que é processado (interpretado) no orquestrador através de um programa denominado de motor (*engine*) de orquestração<sup>5</sup>. Nesse *script* é determinada toda a lógica de execução para uma chamada de um cliente,

---

<sup>5</sup> No decorrer deste texto serão usadas alternadamente, mas com o mesmo sentido, as designações motor e *engine*.



indicando as invocações na ordem correta e o encaminhamento adequado dos resultados. Na Figura 3.5 é apresentado um exemplo de orquestração.

### 3.3.3 A Linguagem para Orquestração – WS-BPEL

O WS-BPEL (*Web Service Business Process Execution Language*) [Arkin, *et al.*, 2007], ou simplesmente BPEL, é uma linguagem para especificar o comportamento de serviços *web* em uma interação de um processo de negócio. A especificação fornece uma linguagem baseada em XML para descrever a lógica de controle necessária para coordenar os serviços *web* participantes do processo. *Scripts* gerados por esta linguagem são interpretados em motores de orquestração, sempre centrados em um dos participantes da composição de serviço considerada. A partir de um motor de orquestração são então ativadas as várias atividades de um processo de negócio, e tratados também os erros que ocorrem nas requisições enviadas.

Com o BPEL podemos descrever um processo de negócio de dois modos distintos:

- **Processo Executável:** as especificações que descrevem este processo determinam o comportamento exato do processo de negócio e o mesmo pode ser executado por um motor de orquestração.
- **Processo Abstrato:** é utilizado para descrever os padrões observáveis de troca de mensagens, permitindo fornecer uma visão para os parceiros de negócio sem se preocupar em mostrar os detalhes internos do processo de negócio. Para um parceiro de negócio saber como interagir com o processo de negócio, basta expor o processo abstrato. As descrições de processos abstratos são mais próximas dos conceitos de coreografia.

A linguagem BPEL então define uma sequência de execução para um processo de negócio (*Business Process*), em uma especificação. Um *script* é gerado desta especificação, para permitir a implantação do fluxo de execução da composição de serviços correspondente.

Os participantes de uma composição de serviços orquestrados são considerados como parceiros (*partners*). Estes parceiros podem invocar outro processo, prover um recurso ao processo em que participam ou ambos. As interfaces dos parceiros são descritas através de documentos WSDL, que determinam as operações permitidas em cada participante. O *script* BPEL define, na execução sequencial do processo, quando e quais

destas operações descritas nestas interfaces serão ativadas. O cliente deve ter também disponível a especificação WSDL do serviço fornecido pela composição.

A Figura 3.6 apresenta o fluxo de execução de um *engine* BPEL. No lado esquerdo, é mostrado o fluxo de um processo de negócio. E no lado direito, está ilustrado este fluxo de execução controlado por um *engine*, que faz o papel do orquestrador do processo de negócio. Ao receber uma requisição do cliente, o *engine* passa a compor a orquestração de acordo com o protocolo de negócio definido pelo *script* correspondente, ordenando a execução de cada componente de forma sequencial ou paralela.

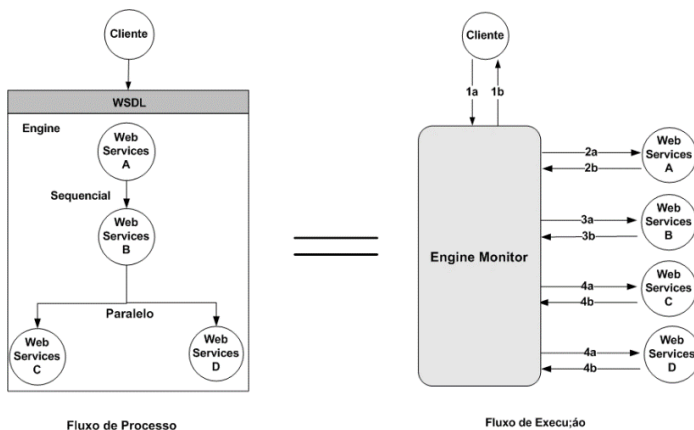


Figura 3.6 – Fluxo de processo e sua execução.

Como qualquer linguagem de programação, o BPEL oferece determinados tipos de construções como *loops*, desvios condicionais, variáveis e atribuições. Este fato acaba possibilitando um tratamento algorítmico do processo de negócio. Na especificação cada instrução é uma atividade e são representadas no documento por elementos XML definidas na estrutura do BPEL. As atividades são divididas em básicas ou estruturadas. Atividades básicas são instruções que definem interações com elementos no processo como, por exemplo, o recebimento (`<receive>`), resposta (`<reply>`) ou invocação (`<invoke>`) de um serviço *web* parceiro. Atividades estruturadas envolvem o estabelecimento da ordem para as atividades nelas aninhadas, ou seja, são estas atividades estruturadas que efetivamente definem a gerência dos fluxos de execução no processo como um todo.

Visando garantir uma maior robustez da BPEL frente a erros ou a transações mal sucedidas, existem construções específicas para tratamento de erros que ocorrem durante a execução de um processo BPEL. Com esta finalidade, algumas funcionalidades são providas em especificações BPEL como, por exemplo, a compensação (*Compensation*) e os tratadores de exceções (*Exception Handler*). A compensação é usada, quando da ocorrência de um erro, para cancelar a atividade que havia sido executada e não foi confirmada devido a este erro. Com isso, esta funcionalidade pode ser usada para reverter operações já executadas na aplicação. Quanto ao tratamento de exceções, de maneira clássica, os tratadores declarados são ativados no momento da sinalização da exceção correspondente. O tratamento pode envolver o redirecionamento de operações para componentes *standby* incluídos na composição.

Um processo de negócio especificado em BPEL tem sua interface baseada no padrão WSDL, troca informações com clientes e serviços *web* através do padrão SOAP e o seu serviço é publicado em um repositório UDDI para fins de busca e consulta. Na prática, um processo de negócio baseado em BPEL se comporta no ponto de vista de seus clientes igual a um serviço *web*.

### 3.4 ORQUESTRAÇÃO DESCENTRALIZADA

Como citado na seção 3.3, composições de serviços *web* baseadas em orquestração possuem uma coordenação centralizada, isto é, um nodo central é responsável pelo recebimento da requisição do cliente, fazendo o tratamento necessário destes dados recebidos e invocando os serviços da composição, de acordo com a especificação definida para o processo de negócio. Além do controle do fluxo das execuções nos serviços da composição, este modo de operação centralizado também determina o acesso a dados intermediários e o envio das respostas aos clientes por parte deste nodo central. Esta centralização dos fluxos de controle e dados cria problemas de desempenho, de segurança e de confiabilidade. Além de um tráfego desnecessário na rede, este tipo de abordagem sofre também de problemas de escalabilidade, pois todo o controle fica centralizado em apenas em um nodo.

Para minimizar esses problemas foram propostos alguns trabalhos que estendem o conceito desta composição de execução centralizada para uma orquestração descentralizada [Binder, *et al.*, 2006], [Chafle, *et al.*, 2004], [Nanda, *et al.*, 2004], [Kyprianou, 2008]. A ideia destas extensões é particionar o fluxo de execução em vários *engines* de orquestração, sendo cada *engine* responsável por uma parte dos fluxos do processo de

negócio. Ou seja, são responsáveis por partes ou subcomposições de serviços da orquestração original. O controle das execuções na composição passa então a ser distribuído.

Deste modo, os *engines* distribuídos devem se comunicar entre si diretamente pela transferência de mensagens, definindo os fluxos de dados e de controle da aplicação distribuída como um todo. Estas comunicações são realizadas de modo não-bloqueante entre os *engines* da composição, sem passar por um coordenador central.

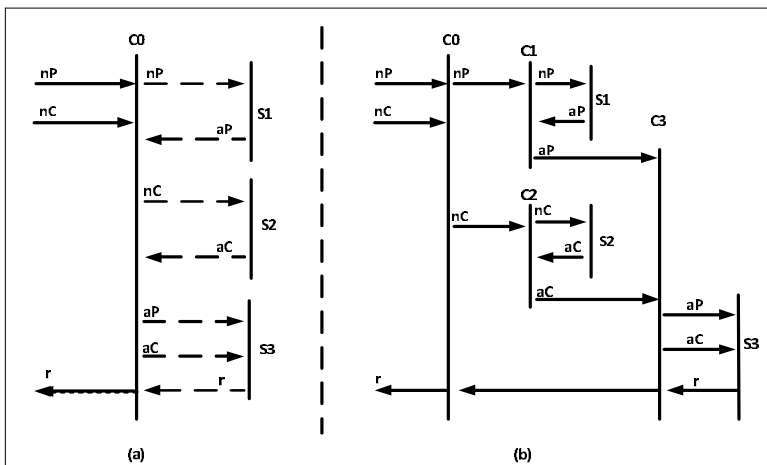


Figura 3.7 – Orquestrações Centralizada (a) e Descentralizada (b).

A Figura 3.7 ilustra exemplos dos modelos centralizado e descentralizado de orquestrações. O cenário apresentado nesta figura é composto por três serviços ( $S_1$ ,  $S_2$ ,  $S_3$ ) e os coordenadores (ou *engines*) responsáveis pela execução do fluxo de controle da aplicação distribuída ( $C_0$ ,  $C_1$ ,  $C_2$ ,  $C_3$ ). O serviço  $S_1$  recebe um valor  $nP$  e tem como resultado  $aP$ . O serviço  $S_2$  precisa de um valor de entrada  $nC$  e retorna como resultado  $aC$  e, por último, o serviço  $S_3$  precisa dos resultados dos serviços anteriores, que são  $aP$  e  $aC$ , e retorna  $r$  como resultado ao cliente da composição.

A Figura 3.7 (a) representa só um orquestrador (um *engine* central, o  $C_0$ ) que atua como um intermediário entre o cliente e todos os serviços. O coordenador central invoca apenas um serviço por vez (de maneira sequencial) até concluir a execução de todos os fluxos da composição e enviar o resultado final ao cliente. Observe que algumas informações ( $aP$  e  $aC$ ) têm passagens desnecessárias pelo coordenador  $C_0$ . O envio destas informações poderia ser feito diretamente para o serviço  $S_3$ . Na

orquestração descentralizada (Figura 3.7 (b)), com o controle particionado e distribuído entre os componentes da composição, o coordenador ( $C_0$ ) envia  $nP$  e  $nC$  de forma paralela para  $S_1$  e  $S_2$ . Os resultados destes serviços são enviados diretamente para o serviço  $S_3$  para ser processado e, ao final, uma resposta de  $S_3$  é retornada ao cliente como resultado da composição, via o controlador  $C_0$ .

### 3.4.1 Trabalhos sobre Orquestração Descentralizada

Em [Chafle *et al*, [2004], é proposto um modelo de orquestração descentralizada onde as possibilidades de controle de fluxo são analisadas e, então, é definido um controle distribuído de pequenos fluxos de execução. Os vários *engines* resultantes da análise são usados para executar os fluxos da composição. Para realizar esta fragmentação do controle, foi definida uma ferramenta que faz esta análise a partir de um código BPEL (BPEL convencional que produz *engines* centralizadores) e que fornece, como resultado, partes de especificações BPEL a serem usadas para gerar os vários *engines* necessários para a orquestração descentralizada.

A ferramenta citada é baseada em um algoritmo que faz o particionamento do código de entrada. Este algoritmo identifica o número de partes possíveis usando as informações dos serviços componentes e do possível fluxo na composição dos mesmos. Com estas informações a especificação BPEL de entrada é dividida entre os componentes de tal forma que os dados a serem transmitidos entre componentes sejam minimizados e o paralelismo entre os serviços é maximizado.

A análise da ferramenta verifica as relações de dependência tanto no fluxo de controle como no de dados da composição de serviços. Por exemplo, se o código BPEL (para *engine* centralizado) incluía operações para tratamento e recuperação de erro, no particionamento de saída desta ferramenta, essas operações serão tratadas de modo a não produzir um bloqueio no sistema. A ideia é que a ferramenta encontre subcomponentes que possam se executar em paralelo de forma completamente independente. Além disso, a ferramenta também fornece uma especificação WSDL de cada parte gerada de código para que estes possam ser invocados como serviços *web*. Ou seja, são tratados como subcomposições dentro da composição total.

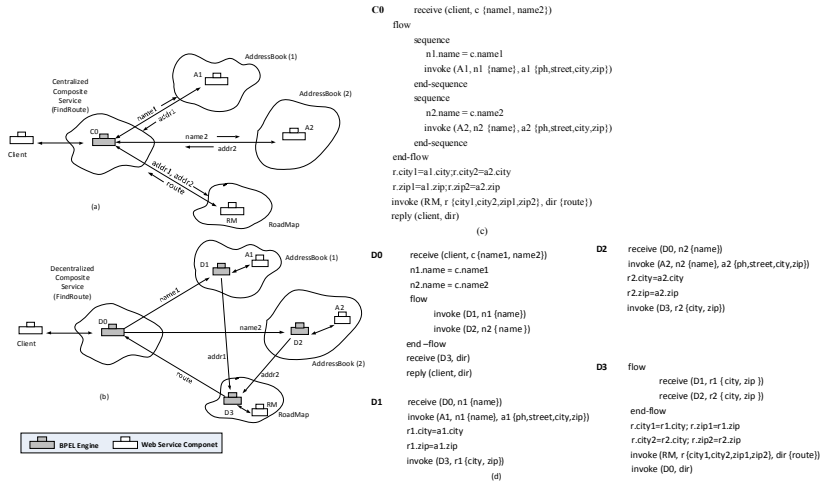


Figura 3.8 – Orquestração centralizada e descentralizada [Chafle, *et al.*, 2004].

A Figura 3.8 ilustra exemplos de um código BPEL de entrada para a ferramenta (para abordagem centralizada) e os códigos resultantes do processo de particionamento. A finalidade da composição de serviços (*FindRoute*) da figura é fornecer nomes e rotas entre endereços. A Figura 3.8(a) representa uma composição de serviços de controle centralizado – C0 é o coordenador central da composição. A sequência no fluxo representado nesta figura coloca C0 recebendo os parâmetros *name1* e *name2* do cliente. Após, C0 envia *name1* para o serviço *addrBook1(1)* que retorna o endereço *addr1* de *name1*. Em paralelo, C0 envia *name2* para *addrBook1(2)* que retorna por sua vez o endereço *addr2* de *name2*. Com os parâmetros *addr1* e *addr2* é extraído o código de área (*zip*) e a cidade (*city*) de cada parâmetro e enviado para o serviço *RoadMap*. Este serviço irá informar um caminho entre os nomes *name1* e *name2* (ou entre os endereços *addr1* e *addr2*) para o cliente.

A Figura 3.8 (b), por sua vez, representa o mesmo exemplo em uma orquestração descentralizada desta composição de serviços. Note que o código da orquestração centralizada (Figura 3.8(a)) é particionado em quatro subcódigos menores – D0, D1, D2 e D3, que ao serem traduzidos produzem quatro *engines* correspondentes.

O algoritmo de particionamento da ferramenta estabelece o encaminhamento dos parâmetros de saída diretamente para os *engines* consumidores destes dados. Desta maneira, o *engine* D0 envia os parâmetros *name1* e *name2* para os *engines* D1 e D2, respectivamente, e

os resultados desta invocação, no caso *addr1* e *addr2*, são encaminhados diretamente ao *engine* D3 (Figura 3.8 (b)). Após a obtenção do caminho entre os endereços *addr1* e *addr2*, D3 retorna o resultado a D0. Note que o *engine* D0 realiza apenas duas invocações, enquanto o *engine* C0 (Figura 3.8 (a)) realiza três invocações. E, para cada uma destas invocações, C0 precisa esperar pelo resultado, aumentando assim o número de mensagens na rede e o tempo consumido no processamento da composição como um todo.

Em [Binder, *et al.*, 2006], por sua vez, é proposto o conceito de *Service Invocation Triggers*, ou simplesmente *Triggers* (gatilhos). Estes *Triggers* têm a função de fazer repasses de parâmetros definindo caminhos diretos entre provedores e consumidores no fluxo de dados, introduzindo assim um controle descentralizado na composição do modelo. Os *Triggers*, nesta função de *buffers* entre provedores e consumidores, atuam então de diversas maneiras sobre os parâmetros:

- Como *buffers*, recebem os parâmetros de entrada para cada um dos serviços, mantendo como uma forma de sincronização dos parâmetros de entrada que podem ser múltiplos e terem origens distintas;
- Disparam a chamada de um serviço após receberem e sincronizarem os parâmetros de entrada;
- Definem o roteamento de cada parâmetro de saída do serviço, já que estes podem ter múltiplos destinos distintos.

Na prática, o modelo de *Triggers* funciona como uma *proxy* especializada. A ação destas *proxies* faz com que o controle de fluxo resultante passe a ter características distribuídas, uma vez que as *Triggers* são encadeadas de maneira a não retornam para um mesmo orquestrador central.

Na Figura 3.9 (a) é apresentada a arquitetura tradicional de orquestração, enquanto que na Figura 3.9 (b) é apresentada a arquitetura baseada em *triggers*, a fim de melhor compará-las. Como mostrado na Figura 3.9 (b), para a otimização da transmissão dos dados o *trigger* atua entre os serviços.

Como desvantagem nesta proposta com *triggers*, o fluxo de execução da composição de serviços é o mais simples possível. Não possui operações como loops e comandos condicionais, etc. Além disso, a aplicação de *triggers* (serviços) não é transparente para o cliente, uma

vez que cada nodo da rede de serviços precisa destes elementos de controle.

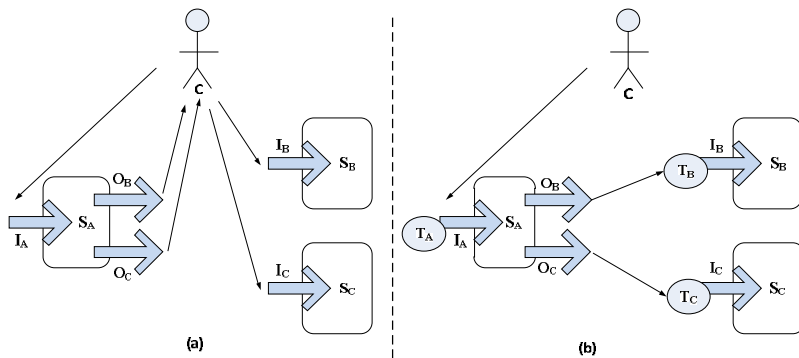


Figura 3.9 – Orquestrações: (a) Centralizada e (b) Descentralizada com *triggers*.

### 3.5 TRABALHOS RELACIONADOS

Atualmente não existe uma especificação *Web Services* padrão que introduza mecanismos ou suportes para a tolerância a faltas nesta tecnologia. Entretanto, é crucial que sistemas distribuídos em ambientes de redes de longa distância sejam tolerantes a faltas e intrusões. Diante disto, vários esforços vêm sendo realizados nestes últimos anos para introduzir nesta tecnologia e na SOA modelos para a detecção, notificação e tratamento de faltas e intrusões. Esta tecnologia de serviços é destinada para redes de grandes distâncias e é necessário que estes mecanismos sejam mantidos de maneira ortogonal às aplicações e transparente ao usuário.

Esta seção tem por objetivo apresentar trabalhos relacionados à implementação de serviços *web* tolerantes a faltas. As soluções encontradas na literatura podem ser divididas em três categorias: 1) serviços *web* que toleram faltas por *crash*, (2) composição de serviços usados na tolerância a faltas por *crash* e (3) serviços *web* que suportam mecanismos de tolerância a faltas bizantinas.

#### 3.5.1 Trabalhos Envolvendo Serviços Web e Faltas por Crash

Em [Liang, *et al.*, 2003] é proposto o primeiro trabalho envolvendo serviços *web* e tolerância a faltas. Os autores introduzem extensões no padrão SOAP permitindo o desenvolvimento da técnica de replicação passiva para tolerar faltas de parada (*crash*). Este modelo realiza também alterações nas especificações WSDL, inserindo desta maneira informações referentes à réplica primária e às réplicas de *backup*. A



utilização de interceptadores na camada SOAP no cliente permite o redirecionamento da requisição para réplicas *backup* em caso de falhas do primário. No servidor, os interceptadores adicionam componentes para registro em logs, detecção de falhas e gerenciamento das réplicas.

Os trabalhos abordados em [Dialani, *et al.*, 2002], [Zhang, *et al.*, 2004], [Townend e Xu, 2005], propõem modelos tolerantes a faltas para serviços *web* implementados e executados sob as especificações de serviços em *grids* [OGSA, 2003]. Em [Dialani, *et al.*, 2002] o objetivo principal da arquitetura proposta é a detecção e a recuperação em situações de faltas. Este modelo não trata a tolerância de falta através da replicação de objetos, mas através de mecanismos de *checkpoint* e *rollback*. Em [Zhang, *et al.*, 2004] é utilizada a técnica de réplica passiva através de mecanismos de notificação providos pela infraestrutura de *grid*. Em [Townend e Xu, 2005] é proposta a implementação de replicação ativa que executa um conjunto de serviços equivalentes (*n-version model*). Esta proposta envolve então aspectos de diversidade de projeto e o uso de diferentes plataformas de execução. O modelo atua sobre as respostas com mecanismos que consideram a frequência das respostas recebidas.

Em [Santos, *et al.*, 2005] é introduzida uma infraestrutura para tolerância a faltas em aplicações na forma de serviços *web*. Essa infraestrutura, denominada FTWEB, é baseada nos conceitos e padrões definidos pela especificação FT-CORBA [OMG 2002]. As réplicas do serviço são organizadas em um único grupo, sendo que as requisições dos consumidores são enviadas para um componente denominado *WSDispatcher*. Esse componente é responsável por gerenciar as réplicas do serviço e enviar as requisições recebidas dos consumidores para essas réplicas. Essa infraestrutura usa as ideias e conceitos presentes no FT-CORBA para ter acesso a objetos CORBA com métodos que são invocados como se fossem serviços *web*. Para realizar essa invocação de métodos em objetos CORBA e a tradução dessas invocações para os protocolos e padrões utilizados em serviços *web*, foram definidos alguns componentes com essa finalidade. Esses componentes são o *WSWrapper* e o *WSClientDriver*. O primeiro é responsável pela integração desses objetos com o restante da infraestrutura, enquanto o segundo componente é usado junto com a aplicação cliente responsável por enviar as requisições desse cliente para o serviço.

Em [Salas, *et al.*, 2006] é proposta uma infraestrutura de serviços denominada *WS-Replication* que visa a alta disponibilidade de serviços *web*. O principal objetivo desta infraestrutura é a replicação de serviços *web* em uma WAN (*Wide Area Network*). A replicação ativa é usada na

organização de um grupo de réplicas do serviço ao longo de uma WAN. A infraestrutura está baseada na definição de dois componentes principais: o componente *WS-Replication* e o componente *WS-Multicast*. O componente *WS-Replication* implementa a gerência das réplicas de serviço com base no modelo replicação ativa (esta gerência envolve aspectos de ordenação e de decisão na resposta a clientes). O componente *WS-Multicast* é fundamental para garantir a disseminação das requisições de serviço para todas as réplicas. Esse componente *WS-Multicast* é construído sobre o protocolo SOAP.

O modelo proposto por [Aghdaie e Tamir, 2002] realiza alterações nos núcleos (*kernels*) de sistemas operacionais e nos servidores *web* para prover tolerância a faltas de *crash* transparente para o cliente. Neste modelo, cada pedido recebido pelo servidor é registrado e enviado também a um servidor *backup*. As alterações realizadas em *kernel* de sistema operacional fornecem uma extensão que agrega ao mesmo um suporte de *multicast confiável*, permitindo desta forma que pedidos enviados ao primário cheguem também a servidores *backup*. As alterações realizadas em servidores *web* permitem a manipulação e geração das respostas para os clientes.

### 3.5.2 Composição de Serviços Web e Tolerância a Faltas por Crash

Em [Tartanoglu, *et al.*, 2003], é descrito o *middleware WSCA (Web Service Composition Action)*, cuja finalidade é a recuperação de erro na execução de uma composição de serviços. A ideia é prever a ocorrência de possíveis exceções em uma composição de serviços e, quando ocorrem, estas devem ser tratadas sem implicar no reinício da execução da composição. Para tanto, é requerida a detecção de falhas em eventos da composição e o uso dessas falhas parciais na sinalização de exceções e de seus tratamentos necessários para a recuperação da composição.

O trabalho abordado em [Dobson, 2006] explora o uso da especificação *WS-BPEL* para a implementação de um serviço tolerante a faltas. Propõe um protótipo, na forma de um processo de negócios, onde um *engine* de orquestração, ao receber as requisições do cliente, gerencia as invocações entre as réplicas do serviço. O *engine* executa uma votação nas respostas recebidas das réplicas e, como consequência desta votação, envia uma resposta ao cliente. Este trabalho menciona *Web Services* como *stateful* no modelo proposto, e assim, as invocações pelo *engine* são realizadas de forma a garantir o determinismo das réplicas, usando requisitos de acordo e ordem sobre as requisições (invocações sobre a máquina de estado).

Em [Laranjeiro e Marco Vieira, 2007], é definido o suporte FTWS que permite aos desenvolvedores especificar serviços *web* alternativos para cada requisição emitida. A técnica proposta utiliza um *proxy* de serviço como intermediário entre o *engine* da composição e cada réplica de serviço. Com isso, quando o *engine* de orquestração faz uma invocação, a requisição é enviada para o *proxy* e através desta, a invocação é concretizada na réplica de serviço correspondente.

Em [Lau, *et al.*, 2008], também é utilizada a especificação WS-BPEL para prover suporte de tolerância a faltas de *crash*. Esta proposta descreve a infraestrutura FTWS-Orch, baseada em uma combinação de técnicas de replicação ativa e passiva. É composta por réplicas primária e *backups* de *engines*. Requisições de cliente são passadas ao primário que gerencia todas as operações de invocações de serviço, de tratamento do erro e de votação sobre respostas. Se a réplica primária falhar, os pedidos são automaticamente transferidos para uma das réplicas *backup*.

### 3.5.3 Trabalhos com Serviços Web e Faltas Bizantinas

Embora a segurança e a confiabilidade sejam requisitos vitais em aplicações críticas, proposições de modelos para arquiteturas orientadas a serviço atendendo a tolerância a falhas maliciosas são ainda relativamente recentes e em número reduzido. Em [Merideth, *et al.*, 2005], [Zhao, 2007] são introduzidos modelos de infraestrutura de serviços cuja preocupação é a tolerância a intrusões (faltas bizantinas) em provedores de serviços *web*. São baseados no algoritmo PBFT de Castro e Liskov [1999] para garantir ordem total das mensagens de aplicação.

O *middleware Thema* introduzido em [Merideth, *et al.*, 2005] foi proposto com a finalidade de prover serviços tolerantes a faltas bizantinas. Este *middleware* é formado por três bibliotecas que se executam fazendo uso do protocolo SOAP. Estas bibliotecas implementam o protocolo PBFT e a interceptação das mensagens de requisição e resposta no modelo.

Em [Zhao, 2007] é proposta a infraestrutura BFT-WS que também tem por objetivo prover serviços tolerantes a faltas bizantinas. A infraestrutura é composta por um conjunto de componentes que são responsáveis pelo gerenciamento e tratamento das requisições trocadas entre as réplicas. Entre esses componentes se destaca o *BFT-WS In Handler*, que se encontra no lado do servidor. Este componente é responsável pela execução do protocolo PBFT, importante para a garantia de ordem total das requisições, pelo gerenciamento na troca de visão, e pela sincronização de estados da replicação Máquina de Estados. O uso

do PBFT no modelo implica na necessidade de no mínimo  $3f + 1$  réplicas, onde  $f$  é o limite para o número de réplicas faltosas no sistema.

### 3.5.4 Resumo dos Trabalhos Relacionados Usando Tecnologia Web Services

Como mostrado na seção 3.4, existe uma limitação no uso da orquestração devido ao tráfego desnecessário na rede e o problema da escalabilidade causado pela centralização do fluxo de controle e dados. Com isso, foram propostos trabalhos que estendem o conceito de orquestração centralizada para uma orquestração descentralizada como mostrado em [Binder, *et al.*, 2006], [Chafle, *et al.*, 2004]. As propostas apresentadas têm como objetivo a distribuição do controle do fluxo de execução em uma composição através do uso de vários *engines* na orquestração.

Outro fato levantado é que, apesar de toda a flexibilidade provida pela tecnologia de Serviços *Web*, ainda não se tem uma especificação ou padronização que atendam requisitos de tolerância a faltas. É neste cenário que na seção 3.5, são apresentados alguns trabalhos que introduzem modelos e mecanismos de tolerância a faltas para arquiteturas orientadas a serviço.

Muitos destes trabalhos limitam suas proposições em soluções para faltas de parada (*crash*). São trabalhos que propõem normalmente o redirecionamento da requisição de um serviço para outro com a mesma interface, diante da falha do primeiro serviço acessado. Os trabalhos que consideram os estados das aplicações normalmente baseiam suas técnicas de tolerância a faltas em modelos de replicações passivas, ativas ou semiativa.

A utilização de composição de serviços na tolerância a faltas é focada também em modelo de falhas por *crash*. São trabalhos baseado em um modelo que usa a composição de serviços, normalmente orquestração centralizada, para implementar serviços tolerantes a faltas.

São poucos os trabalhos envolvendo tolerância a intrusões em serviços *web* e em composição de serviços (faltas bizantinas). Isto se deve principalmente à complexidade envolvida na tolerância a malícia. Mas, os ataques e intrusões em ambientes abertos e as consequentes corrupções de serviços não podem ser ignorados. Os trabalhos existentes são baseados no algoritmo PBFT de Castro e Liskov [1999] onde, para alcançar o acordo entre as réplicas de serviço, são necessárias no mínimo  $3f + 1$  réplicas. Além de necessitar um número maior de réplicas, existe ainda o problema do número de passos de comunicação necessário para se chegar ao acordo. A Tabela 3.1 representa uma comparação dos

trabalhos discutidos neste capítulo onde procuramos evidenciar as principais características de cada uma destas propostas.

Tabela 3.1 – Comparação dos trabalhos discutidos.

	Serviços Web e Falhas por Crash							
	<i>Liang, et al</i>	<i>Djalani, et al</i>	<i>Zhang, et al</i>	<i>Townend e Xu</i>	<i>Santos, et al</i>	<i>Salas, et al</i>	<i>Tamir, et al</i>	<i>Ye, et al</i>
Técnica de Réplicação	passiva	-	passiva	passiva	vários	ativa	seme-ativa	ativa
Ordenação das Mensagens	sim	-	-	não	sim	sim	não	sim
Deteção de Falhas	sim	sim	sim	não	sim	sim	sim	não
Recuperação de Falhas	sim	-	sim	-	sim	-	-	-

	Composição de Serviços Web e Falhas por Crash			
	<i>Tartanoglu, et al.</i>	<i>Dobson, et al</i>	<i>Laranjeir, et alo</i>	<i>Lau, et al</i>
Técnica de Réplicação	-	passivo	ativa	ativa
Ordenação das Mensagens	-	-	sim	sim
Deteção de Falhas	sim	sim	-	não
Recuperação de Falhas	sim	sim	-	não

	Serviços Web e Falhas Bizantinas	
	<i>Merideth, et al</i>	<i>Zhao</i>
Número de Réplicas	$3f+1$	$3f+1$
Algoritmo	PBFT	PBFT
Passo de Comunicação	4	4
Tipo de Ambiente	LAN	LAN

### 3.6 CONCLUSÃO DO CAPÍTULO

A Arquitetura Orientada a Serviços é uma evolução de tecnologias de sistemas distribuídos, como DCOM, CORBA e Java RMI. Provê uma camada de abstração que permite que aplicações existentes possam ser encapsuladas e tratadas como serviços. Com isso, não existe a necessidade de reescrever aplicações existentes (legadas), na forma de serviços, para atuarem em um processo de negócio de uma organização.

Desta maneira, o foco principal da SOA é a integração de aplicações existentes e não na implementação destas. Com isso, este tipo de arquitetura provê uma transparência dos serviços nas suas implementações, reduzindo o impacto quando ocorre qualquer modificação na implementação ou na própria infraestrutura que suporta conceitos SOA. Assim, a integração de aplicações dentro de uma ou mais organizações em redes WANs se torna mais fácil, mas o gerenciamento desta integração de serviços nem sempre é simples.

É neste contexto que a composição de serviços *web* se insere propondo uma solução para o gerenciamento destas integrações de serviços em processos de negócio. A composição de serviços, através das abordagens de orquestração ou da coreografia, permite de uma forma mais concisa e segura a especificação das interações entre serviços de um processo de negócio, garantindo a interoperabilidade e a independência de plataformas entre as mais diversas aplicações, sistemas operacionais e equipamento.

A partir das análises apresentadas neste capítulo podemos perceber que existem possibilidades de se explorar no contexto de tolerância a intrusões as abordagens e conceitos de composição de serviços.

## 4 SERVIÇO TOLERANTE A INTRUSÕES

Os *Service Providers* estão disseminados na Internet e fazem parte de nossas rotinas diárias. Esta profusão de serviços vem também acompanhada de preocupações. São preocupações normalmente ligadas à segurança. Embora toda a evolução experimentada pelas tecnologias de segurança, as invasões de provedores de serviço são fatos frequentes na Internet. A dificuldade em lidar com este problema é devido à complexidade sempre crescente dos suportes para aplicações disponíveis na rede mundial. Por mais cuidado que um projetista tenha no desenvolvimento destes sistemas, por mais que se acerque de técnicas de prevenção, vulnerabilidades sempre estarão presentes nestes sistemas e aplicações.

Na dificuldade de evitar acessos ilegais, muitas abordagens têm sido apresentadas na literatura no sentido de conter ou restringir as possíveis intrusões que ocorrem em sistemas de ambientes abertos. Estas técnicas compõem o que usualmente é chamado de tolerância a intrusões [Correia, 2005] [Fraga e Powell, 1985]. A ideia, neste caso, é manter o sistema distribuído evoluindo com comportamento correto mesmo diante de alguns componentes já corrompidos por intrusões. A ação maliciosa destes componentes invadidos não deve afetar o sistema como um todo. Na literatura de *Dependability* [Laprie, 1995], a noção de faltas bizantinas (ou faltas maliciosas) é usada para descrever o comportamento de subcomponentes comprometidos de um sistema. E as soluções para estas corrupções parciais de um sistema envolvem o uso de técnica de replicação ativa, também conhecidas como replicação Máquina de Estado (ME) [Schneider, 1990].

Em [Zhao, 2008] e [Merideth, *et al.*, 2005] são introduzidos modelos de infraestrutura de serviços cuja preocupação é a tolerância a intrusões (faltas bizantinas) em provedores de serviços *web*. Estes modelos são construídos tomando como base replicações ME.

A motivação principal do texto deste capítulo é apresentar uma alternativa aos problemas citados acima, desenvolvida no escopo desta tese. Esta nossa experiência introduz uma infraestrutura que faz uso extensivo de virtualização na implementação de ME em seu suporte de tolerância a intrusões (faltas maliciosas ou bizantinas) para serviços *web*. O modelo, com o uso da virtualização, define um elemento confiável que determina a separação entre faltas maliciosas e as usuais “*crashes*” de máquinas físicas, e ainda a simplificação de nossos algoritmos. As réplicas de serviço construídas sobre máquinas virtuais ficam envolvidas exclusivamente com protocolos BFT (os mais custosos) limitados,

portanto, a trocas via memória compartilhada. A infraestrutura de serviços implementada a partir de replicação ME faz uso também de conceitos de Orquestração Descentralizada e de padrões de *Web Services*, permitindo que provedores de serviço possam ser construídos de modo a tolerar intrusões sem ter que lançar mão de clusters que é a solução usual encontrada na prática.

Neste capítulo, será abordado especificamente nossa proposta de trabalho executado em nível local. Inicialmente, será apresentado o modelo de sistema adotado em nosso trabalho, em seguida serão apresentados os protocolos de consenso executados em um ambiente virtualizado em uma rede local. Por último, serão apresentados os experimentos realizados visando verificar a viabilidade prática do modelo e protocolos propostos.

#### 4.1 MODELO DE SISTEMA

Nosso modelo é composto por dois grupos de processos que interagem através de comunicação via rede: os servidores e os clientes. O conjunto de servidores,  $S = \{S_0, S_1, \dots, S_{n-1}\}$ , também chamados de réplicas de execução, é formado por  $n$  processos com a mesma funcionalidade, concretizando a replicação Máquinas Estados. Os clientes no sistema,  $C = \{C_0, C_1, \dots\}$ , são processos que enviam requisições para os servidores através da rede.

No modelo, processos (clientes e servidores) são classificados como corretos ou faltosos. Processos corretos se comportam segundo as especificações do algoritmo, enquanto processos faltosos se comporta de maneira arbitrária desviando de suas especificações, podendo como tal, omitir mensagens, parar de responder, modificar o conteúdo das mensagens e fazer conluio com outros faltosos no sentido de dificultar a evolução correta da aplicação. Este tipo de comportamento é associado na literatura a faltas arbitrárias, bizantinas ou maliciosas (intrusões).

Neste modelo de sistema assumimos um comportamento *parcialmente síncrono* [Dwork, *et al.*, 1988], ou seja, as interações entre processos clientes e o serviço no sistema distribuído admitem intervalos de assincronismo intercalados por períodos estáveis (síncronos). Estes períodos síncronos permitem a terminação destas comunicações em



prazos de tempo definidos, porém não conhecidos. No modelo, os canais de comunicação são também assumidos como *confiáveis*<sup>6</sup>.

Neste modelo, separamos também as réplicas da ME que executam as requisições do cliente, do mecanismo de acordo que denominamos de *Agreement Service (AS)*. Além disso, foi definido um mecanismo que permite a comunicação entre réplicas e o *Agreement Service* e que é concretizado através de uma abstração de memória compartilhada (*Shared Memory*).

A correção nos acessos de memória compartilhada necessita de garantias de *liveness*. Em [Fernandez, et al., 2007] é introduzida a propriedade *AWBI* que define que um processo não se comporta indefinidamente como uma execução assíncrona. Como consequência, a memória compartilhada por processos sempre permitirá acessos em tempo limitado, mas não necessariamente conhecido. Assumimos com isto que a memória compartilhada do modelo proposto também possui um comportamento parcialmente síncrono.

A Figura 4.1 ilustra uma visão geral em relação às interações de cliente, réplicas e o *AS*.

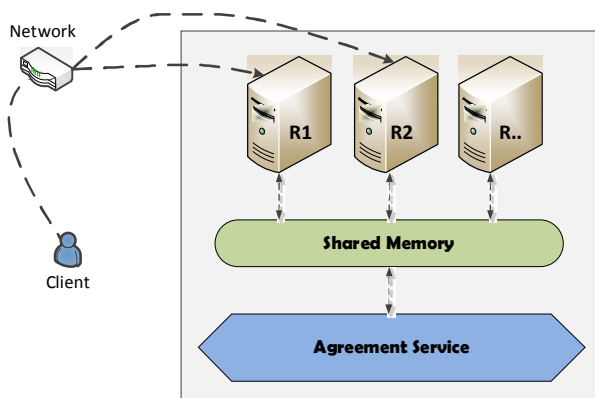


Figura 4.1 – Modelo de interação entre os processos.

<sup>6</sup> Canais de comunicação são ditos *confiáveis* quando mensagens não são criadas ou duplicadas e cada mensagem enviada por um processo  $P_i$  terminará por ser recebida pelo receptor  $P_j$  se o emissor ( $P_i$ ) não for faltoso ou malicioso [Raynal, 2005]

No modelo usamos a técnica de criptografia assimétrica para garantir a autenticidade das mensagens trocadas entre processos. Clientes e o *AS* são os componentes do modelo que possuem seus pares de chaves assimétricas (pública e privada), sendo a chave privada conhecida apenas pelo seu proprietário (cliente ou *AS*). As chaves públicas de todos os clientes do sistema e do *AS* são disponíveis na forma de certificados para todos os participantes do sistema. A verificação de assinaturas em mensagens trocadas envolve também o envio destes certificados de chave pública dos assinantes<sup>7</sup>. As assinaturas envolvem também *hashes*, *nonces* e outros mecanismos usuais adotados em assinaturas digitais, mas que não aparecem explicitamente nos algoritmos apresentados neste trabalho.

Assumimos o uso da técnica de diversidade de software [Avizienis e Kelly, 1984] [Littlewood e Strigini, 2004] para a construção do arcabouço do modelo que será executado pelas réplicas. O objetivo é permitir uma maior independência de falhas, de modo que, as réplicas de execução são implementadas em sistemas operacionais distintos e que tais sistemas não compartilhem das mesmas vulnerabilidades. Deste modo, em uma eventual corrupção de uma réplica, através de uma intrusão, não implica diretamente no comportamento malicioso das demais réplicas do sistema. A diversidade na construção das réplicas diminui a probabilidade que tenhamos a mesma vulnerabilidade se reproduzindo em todas as réplicas e, com isto, diminui as chances de intrusões semelhantes em todos os servidores que compõem o sistema.

#### 4.1.1 Abordagem de Replicação do Modelo

As soluções para problemas de BFT (“*Byzantine Fault Tolerance*”) sempre tomam como base a técnica de replicação Máquina de Estado (ME) [Schneider, 1990]. Em nossa proposta, na submissão de uma requisição de cliente, a implementação desta técnica inicialmente, envolve um procedimento otimista com o uso de somente  $f + 1$  réplicas respondendo a requisição (assumimos inicialmente a inexistência de falhas e intrusões entre as réplicas do serviço). Porém, na detecção de um eventual desacordo entre as  $f + 1$  respostas novas  $f$  réplicas são ativadas, elevando o número de respostas de réplicas para  $2f + 1$ . O valor de  $f$

---

<sup>7</sup>A certificação das chaves públicas das entidades participantes não necessariamente precisa envolver *CAs* (Autoridades Certificadoras) oficiais e reconhecidas. Estes certificados podem ser emitidos por entidades como o comitê gestor (ou o administrador) do serviço replicado e que passa ser entendida pelos participantes como entidade confiável.

corresponde ao limite máximo de faltas ou intrusões admitidas no sistema distribuído de modo a não comprometer a correção do mesmo.

A nossa abordagem de BFT envolve também a identificação e substituição de réplicas corrompidas que provocaram a inconsistência (divergência de respostas) nas trocas com o cliente.

#### 4.1.2 Virtualização

Conforme já citado, o objetivo deste trabalho é dar suporte para serviços de modo que os mesmos sobrevivam a intrusões. No modelo de replicação, as réplicas invadidas e corrompidas não devem comprometer o atendimento das requisições pelo serviço replicado. Para atender estes objetivos, a nossa abordagem de BFT é então construída através do uso da tecnologia de virtualização: os servidores são implementados e executados em separado, usando diferentes máquinas virtuais.

Como nos fixamos no contexto de faltas maliciosas envolvendo réplicas de serviço comprometidas por intrusões, definimos o mapeamento das diversas máquinas virtuais que executam as réplicas em uma única máquina física servidora. Este enfoque implica em um modelo híbrido de faltas que separa os diferentes tipos falhas. Ou seja, em nível de máquinas virtuais tratamos exclusivamente com faltas bizantinas (maliciosas). O problema de faltas de parada (*crash faults*) na máquina física servidora pode ser tratado com protocolos mais brandos em nível de sistema distribuído que será o objetivo do próximo capítulo (capítulo 5). Neste capítulo, nos limitamos à discussão de nossas soluções para faltas maliciosas nas máquinas virtuais.

O uso desta tecnologia de virtualização na implementação da nossa abordagem de BFT também oferece facilidades para o gerenciamento das diferentes réplicas do modelo. Como colocamos anteriormente, novas réplicas podem ser lançadas quando do desacordo das  $f + 1$  respostas a uma requisição de cliente e, também, réplicas identificadas como corrompidas devem ser excluídas do modelo. Nesta tecnologia, máquinas virtuais podem ser facilmente iniciadas ou revogadas quando necessário.

Diante do exposto, o nosso modelo acaba se compondo no seu ambiente de virtualização em dois níveis de estratificação: o primeiro, o *sistema hospedeiro* que é representado por um monitor de máquina virtual (*VMM - Virtual Machine Monitor*), e o *sistema convidado* que em nosso caso é formado pelos servidores executando sobre as máquinas virtuais dispostas sobre o servidor físico. Dependendo do contexto, essas máquinas virtuais também são denominadas *domínios* [XEN, 2009].

Em nosso modelo, assumimos que o *VMM* pode apresentar vulnerabilidades, porém estas não podem ser exploradas externamente via

rede ou pelas máquinas virtuais locais. Para garantir essa suposição, confiamos que o *VMM* forneça o isolamento adequado para a execução segura das máquinas virtuais. Esta é uma das premissas sobre a tecnologia de virtualização [Chun, *et al.*, 2008] [Sahoo, *et al.*, 2010]. Além disso, é necessário que o *VMM* seja externamente inacessível, ou seja, entidades externas não podem ter acesso direto a estes níveis do sistema. Para isso, o *VMM* desabilita os protocolos de rede para estes níveis. Mantendo o sistema hospedeiro inacessível, nada impede que o *VMM* garanta o acesso das máquinas virtuais à rede. Desta forma, os clientes terão, via rede, somente a visibilidade das máquinas virtuais e suas réplicas de execução de modo a enviar suas requisições.

### 4.1.3 Memória Compartilhada

No modelo, fazemos o uso de uma abstração de memória compartilhada (*shared mem*) que assumimos como um conjunto de registradores que armazenam valores e podem ser acessados por dois tipos de operações: escrita e leitura. É através dessas operações que os processos (réplicas e o *Agreement Service*) utilizando estes registradores para se comunicar [Cachin, *et al.*, 2011]. A execução e o gerenciamento desta memória compartilhada são realizados pelo *VMM*.

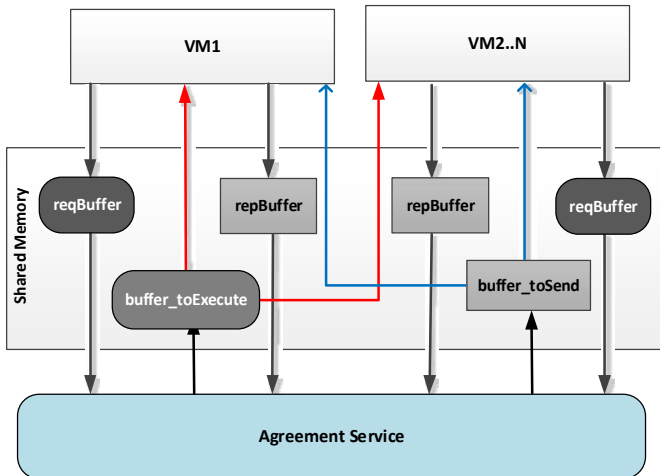


Figura 4.2 – Mecanismos de memória compartilhada.

Além disso, são definidas políticas de acesso para que as réplicas de execução tenham acesso de leitura e escrita em áreas próprias desta memória, isto é, cada máquina virtual (e réplica correspondente) só pode

escrever em sua área particular (nos *buffers reqBuffer<sub>Rj</sub>* e *repBuffer<sub>Rj</sub>*) no envio de suas mensagens para o *AS*. Já as leituras destas máquinas virtuais (réplicas) concorrem sobre dois *buffers* (*buffers* de leitura: *buffer\_toExecute* e *buffer\_toSend*) que tem como escritor um serviço de suporte: o *Agreement Service*. As políticas definidas para esta memória compartilhada são necessárias para garantir a integridade das requisições de clientes já ordenadas e das respostas correspondentes, uma vez que, réplicas maliciosas não podem modificar o conteúdo escritos em registradores particulares de outros processos. A Figura 4.2 ilustra os mecanismos de memória compartilhada considerando seus aspectos funcionais na comunicação.

#### 4.1.4 Agreement Service

Um componente fundamental no nosso modelo é o *Agreement Service* (AS). Este serviço de suporte é responsável pela ordenação das requisições, verificação das respostas das réplicas de execução, geração e assinatura da resposta que será enviada para o cliente. Além disso, coordena a validação dos *checkpoints* feitos pelas réplicas de execução e a ativação de novas réplicas de serviço em caso de desacordo entre as réplicas nas respostas ou *checkpoints*. Ou seja, este componente trata de aspectos não funcionais, de forma que pode ser usado em qualquer tipo de aplicação sem modificações.

Este componente também pode ser considerado como uma abordagem de separação dos serviços de ordenação e de execução da aplicação, a exemplo do trabalho descrito em [Yin, *et al.*, 2003]. Desta forma, as réplicas de serviço que executam a aplicação, ficam expostas aos clientes via rede, enquanto o *Agreement Service* trata da ordenação total das requisições. Por ser uma parte crítica em nossa proposta, assumimos o componente como confiável e é mantido isolado no modelo.

Para garantir esta premissa de um componente confiável e do isolamento entre as máquinas virtuais e o *VMM*, o *Agreement Service* é construído na mesma camada de virtualização onde é executado o *VMM*. Esta escolha se baseia no fato de que a tecnologia de virtualização deve proporcionar o isolamento em relação aos domínios onde são executadas as máquinas virtuais e também o *VMM* [Reiser e Kapitza, 2009] [Reiser, *et al.*, 2006] [Stoess e Götz, 2004]. Outra vantagem em executar nessa camada é o gerenciamento do funcionamento das máquinas virtuais, uma vez que, durante a execução do protocolo novas réplicas podem ser ativadas e desligadas em caso de desacordo nas respostas, e o *AS* deve ser o agente destas reconfigurações.

## 4.2 PROTOCOLO *ITVM*

Como descrito anteriormente, nesta primeira parte será descrito o protocolo de consenso, denominado *ITVM* (*Intrusion Tolerance using Virtual Machines*) que é executado em um ambiente virtualizado em uma rede local. Inicialmente, é apresentado o funcionamento do protocolo sem presença de falhas (modo otimista). Depois, na sequência apresentamos a evolução do protocolo na presença de falhas maliciosas. Serão descritos também os detalhes dos algoritmos de cada uma das entidades envolvidas no protocolo.

### 4.2.1 Execução Sem-Falha do Protocolo *ITVM*

A execução do protocolo no modo otimista ocorre quando não se tem a presença de réplicas maliciosas no sistema, neste caso o protocolo *ITVM* executa as suas operações em modo normal onde são necessárias somente  $f + 1$  réplicas para a execução do protocolo. A Figura 4.3 ilustra a execução otimista do protocolo *ITVM*.

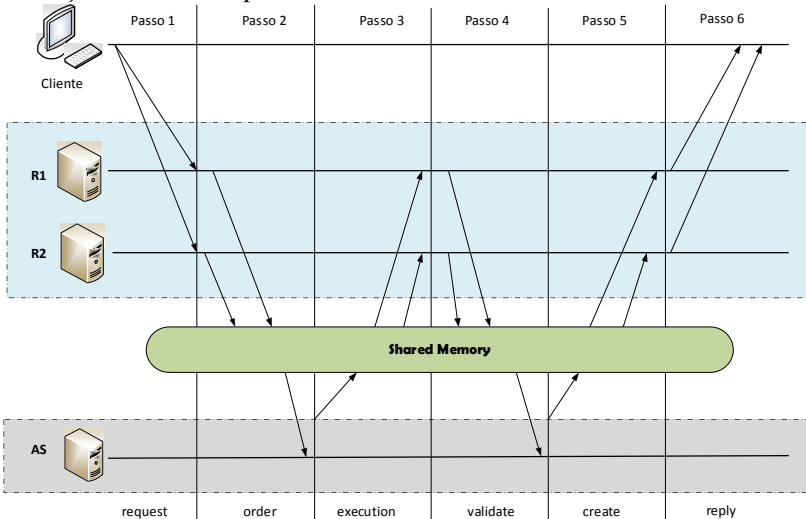


Figura 4.3 – Execução do protocolo sem falha.

No exemplo ilustrado na figura, o limite de falhas é definido como  $f = 1$ . Como consequência, a configuração do servidor replicado, no modo otimista, é de  $f + 1 = 2$  réplicas. O protocolo inicia com o cliente *C* enviando uma requisição (mensagem *request*) para todas as réplicas de execução (R1 e R2, no passo 1 da Figura 4.3). Após o recebimento, as réplicas de execução irão escrever a requisição na memória compartilhada

(*Shared Memory*) (passo 2). O passo seguinte, na memória compartilhada, corresponde à leitura desta requisição pelo *Agreement Service*. O AS deve ordenar esta requisição e através de uma escrita colocar a mesma novamente na memória compartilhada (passo 3).

O próximo passo da execução é a leitura da memória compartilhada pelas réplicas de execução. Ao lerem a requisição ordenada pelo *Agreement Service*, as réplicas executam a operação enviada pelo cliente e um resultado (resposta) da execução é gerado e inserido na memória compartilhada (passo 4). Inserida as respostas na memória compartilhada, o *Agreement Service* irá verificar se existem  $f+1$  mensagens de respostas iguais para a requisição. Em caso afirmativo, é criada uma mensagem de resposta assinada e disponibilizada na memória compartilhada (passo 5). As réplicas devem então ler esta resposta assinada e enviar a mesma para o cliente (passo 6).

O término da instância do protocolo ocorre quando o cliente recebe pelo menos uma resposta vinda de qualquer réplica de execução com uma assinatura válida e efetuada pelo *Agreement Service*.

#### **4.2.2 Execução do Protocolo ITVM na Presença de Réplicas Maliciosas**

Nesta seção, é descrita a execução do protocolo quando se tem réplicas apresentado um comportamento malicioso durante a execução do protocolo. A inconsistência ou desacordo podem ocorrer e devem ser tratados de modo a não comprometer o funcionamento do protocolo. A Figura 4.4 ilustra a execução do protocolo e a estratégia utilizada para tratar a presença de réplicas maliciosas no sistema.

Os passos do 1 ao 3 da execução do protocolo são os mesmos executados no caso quando não se tem réplicas maliciosas. Mesmo que a réplica maliciosa se negue a entregar a requisição no passo 1, a requisição é ainda entregue pela réplica correta. E como esta requisição entregue vem assinada e com o certificado do cliente, a evolução do protocolo se mantém. No entanto, o comportamento malicioso pode se manifestar após a leitura da requisição ordenada e a execução da operação. Ou seja, a réplica maliciosa R1 escreve um valor (resposta) diferente do esperado pela execução do algoritmo. Com isto, a réplica maliciosa causa um desacordo das réplicas de execução sobre as respostas que foram inseridas na memória compartilhada (passo 4 da Figura 4.4).

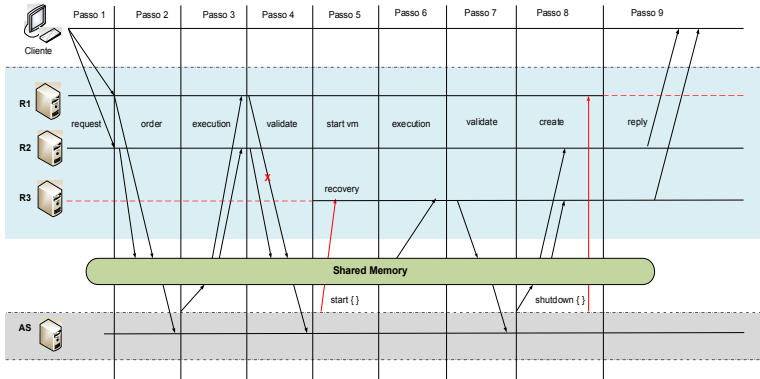


Figura 4.4 – Execução do protocolo na presença de réplica maliciosa.

O problema causado devido a esse desacordo, detectado pelo *Agreement Service*, é que com apenas  $f + 1$  respostas inseridas na memória pelas réplicas não é possível gerar uma resposta correta e assinada para ser enviada para o cliente. Uma vez que, para gerar uma resposta correta é necessário que se tenha pelo menos  $f + 1$  respostas iguais para uma mesma requisição.

Quando detectado esse desacordo nas diferentes respostas obtidas da memória compartilhada, o *Agreement Service* inicializa  $f$  novas réplicas (R3), de modo a aumentar o número de réplicas de execução para  $2f + 1$ . Com esse aumento de réplicas é possível resolver o problema do desacordo e responder a requisição enviada pelo cliente. As novas réplicas inicializadas deverão atualizar o seu estado do sistema, obtendo uma cópia dos dados da aplicação que está sendo executada (passo 5 e 6). Após essa recuperação do estado da aplicação, a nova réplica (R3) irá executar a requisição onde houve o desacordo e escrever a resposta correspondente na memória compartilhada (passo 7). A atualização da nova réplica será discutida mais detalhadamente na seção 4.2.4.

Após o envio da resposta pela nova réplica, o *Agreement Service* agora contendo  $2f + 1$  respostas (R1, R2 e R3), pode identificar a resposta correta, tomando como base as  $f + 1$  (quórum mínimo) respostas iguais das réplicas corretas e com isso validar e gerar uma resposta assinada (passo 8). Esta resposta assinada é inserida normalmente na memória para ser lida e enviada pelas réplicas ao cliente (passo 9).

Com a identificação da resposta correta, o *Agreement Service* inicia o processo de retirada das réplicas que causaram o desacordo. Isto é, as réplicas que produziram respostas divergentes na memória compartilhada são identificadas e suas máquinas virtuais são desligadas.



Com isso, o sistema retorna para o quórum de  $f + 1$  réplicas de execução no protocolo (passo 8).

### 4.2.3 Descrição Detalhada do Algoritmo

O comportamento descrito no algoritmo inicia quando o cliente  $C_i$  envia uma requisição de serviço (operação  $op$ ) às réplicas visíveis via rede em suas máquinas virtuais (linha 3 do algoritmo 1). A montagem da mensagem de requisição ( $req_i$ ) associa um número de sequência ( $n_c$ ) (linha 4). Este número de sequência ao ser concatenado ao identificador do cliente ( $id_{ci}$ ) forma o identificador de mensagem ( $id_m$  linha 5). Completa a montagem da mensagem um campo para a operação desejada ( $op$ ). Na linha 6, parte dos campos da mensagem é assinada com a chave privada do cliente ( $D_{ci}$ ).

---

**Algoritmo 1.** Interações do Cliente  $C_i$  com o Serviço Replicado

---

{Cliente  $C_i$ }

**Init**

- |  |  |
|--|--|
| <ol style="list-style-type: none"> <li>1. <math>n_c \leftarrow 0</math>;</li> <li>2. <math>prazoClient \leftarrow initialValue</math> ;</li> <li>3. <b>upon requestforOperation</b> (<math>op</math>):</li> <li>4. <math>n_c \leftarrow n_c + 1</math>;</li> <li>5. <math>id_m \leftarrow \langle id_{ci}, n_c \rangle</math>;</li> <li>6. <math>Sign_{C_i} \leftarrow Signature(\langle REQUEST, id_m, op \rangle, D_{C_i})</math>;</li> <li>7. <math>req_i \leftarrow \langle REQUEST, id_m, op, Sign_{C_i}, Cert_{C_i} \rangle</math>;</li> <li>8. <b>send</b> <math>req_i</math> <b>to</b> all <math>R_j</math>;</li> <li>9. <math>\delta_{req\_nc.time} \leftarrow timer()</math> ;</li> <li>10. <math>\delta_{req\_nc.id} \leftarrow id_m</math>;</li> </ol> | <ol style="list-style-type: none"> <li>11. <b>upon receive</b> (<math>reply_k</math>) <b>from</b> <math>R_{ej}</math> <b>do</b></li> <li>12.   <b>if</b> <math>verifySign(reply_k.Sign_{AS}, reply_k.Cert_{AS})</math> <b>then</b></li> <li>13.     <b>if</b> <math>\delta_{req\_nc.id} = reply_k.id_m</math> <b>then</b></li> <li>14.       <math>\delta_{req\_nc.time} \leftarrow \perp</math>;</li> <li>15.     <b>end if</b></li> <li>16.     <b>delivery</b> (<math>reply_k.resp</math>);</li> <li>17.   <b>end if</b></li> <li>18. <b>upon</b> (<math>timer() - \delta_{req\_nc.time} \geq prazoClient</math>) <b>at</b> <math>C_i</math> <b>do</b></li> <li>19.   <b>send</b> <math>req_k</math> <b>to</b> all <math>R_j</math>;</li> <li>20.   <math>prazoClient \leftarrow Estimate(prazoClient)</math>;</li> <li>21.   <math>\delta_{req\_nc.time} \leftarrow timer()</math>;</li> </ol> |
|--|--|
- 

A mensagem é completada na linha 7, com a adição da assinatura da mesma e do certificado de chave pública do cliente para verificações posteriores ( $Cert_{C_i}$ ). O envio de  $req_i$  é feito na linha 8 e é dirigido a todas as  $f + 1$  réplicas visíveis do serviço. Nas linhas 9-10, uma estrutura de *timeout* é criada ( $\delta_{req\_nc}$ ), tendo como prazo de espera por respostas, o valor definido por  $prazoClient$ . O tempo de envio ( $\delta_{req\_nc.time} \leftarrow timer()$ ) e o identificador da requisição ( $\delta_{req\_nc.id} \leftarrow id_m$ ) são registrados nesta estrutura de *timeout*.

No algoritmo 1, também são encontrados duas *threads* que representam a recepção da resposta do serviço replicado (linhas 11 – 17) e o tratamento para exceções de respostas que não chegam no prazo definido (linhas 18 – 21). No caso da recepção da resposta ( $reply_k$ ), é feita uma verificação da assinatura da mesma na linha 12, usando o certificado da  $AS_j$  (certificado de quem assina a mensagem de resposta do cliente). Estando correta a verificação, o *timeout* é desativado nas linhas 13-15, a

partir da identificação da mensagem ( $\delta_{req\_nc.id} = reply_k.id_m$ ). A mensagem é então entregue à aplicação cliente (linha 16).

Caso o cliente não receba uma resposta dentro do prazo de tempo definido (*prazoClient*) um tratador correspondente é ativado (linhas 18 – 21). Este tratador retransmite a requisição para a replicação ME (linha 19). O prazo de espera é incrementado de um  $\Delta$  unidades de tempo, através da função *Estimate()* (linha 20) e o temporizador é reativado (linhas 20 e 21). O aumento do prazo leva em consideração as características de ambiente assíncrono definido no modelo do sistema.

O algoritmo 2 representa as trocas quando a requisição do cliente é recebida pelas réplicas de execução e o *Agreement Service* (AS), onde inicialmente é feita uma verificação em seu *cache* se a requisição ( $req_k$ ) já foi executada. Se a resposta for encontrada, a réplica prontamente a envia para o cliente (linhas 2 e 3). Caso a resposta para a requisição não tenha sido encontrada em seu *cache*, as réplicas inserem a requisição  $req_k$  na memória compartilhada (*shared\_mem*), em seus respectivos *buffers*  $reqBuffer_{R_j}$  (linha 5) para que seja definida uma ordem global para a execução da mesma na replicação ME.

Quando notificado da presença de novas requisições na memória o *Agreement Service* irá ler as requisições uma a uma nos *buffers*  $reqBuffer_{R_j}$  (linha 25) de cada uma das réplicas de execução. Em seguida é feita uma verificação para saber se tal requisição já foi ordenada (linha 26). Caso a requisição já tenha sido ordenada, a mesma é descartada (linha 32). Caso contrário, a mensagem é inserida na memória compartilhada no *buffer* *buffer\_toExecute* e a requisição é armazenada na lista de ordenadas (*orderedList*) (linhas 27 e 28), definindo uma sequência para a execução da requisição pelas réplicas. Após a inserção da requisição na memória, um temporizador interno é iniciado, definindo o prazo máximo ( $timeout_k$ ) para a chegada de  $f+1$  respostas iguais para esta requisição (linha 29-30).

Como a escrita da linha 27 no *buffer* *buffer\_toExecute* somente pode ser feita pelo *Agreement Service*, a posição ocupada na memória compartilhada pelas mensagens, considerando suas inserções, vai determinar a ordem de sequência das requisições. Desta forma, as leituras das requisições armazenadas neste *buffer*, sendo respeitada uma ordem FIFO (*First-In-First-Out*), garante que todas as réplicas irão ler e executar as requisições na mesma ordem.

Algoritmo 2. Interações da réplica com o *Agreement Service*


---

```

{ Réplica  $R_j$  }
Var:
1.   $replycache = \emptyset$ ;
Init:
1.  upon receive  $req_k$  at  $R_j$ 
2.  if  $replycache.exists(req)$  then
3.    send  $reply_{ij}$  to  $C_i$ ;
4.  else
5.     $shared\_mem.repBuffer_{R_j}.write(req_i)$ ;
6.  end if
7.  upon ReqSignal at  $R_j$  do
8.     $req\_toExec \leftarrow shared\_mem.buffer\_toExecute.read()$ ;
9.    if VerifySignal( $req\_toExec.Sig_n, req\_toExec.Cert$ ) then
10.      $resp_k \leftarrow thread.execute(req\_toExec.op)$ ;
11.      $rep_k \leftarrow \langle RESPONSE, id_{R_j}, id_{m_r}, resp_k \rangle$ ;
12.      $shared\_mem.repBuffer_{R_j}.write(rep_k)$ ;
13.   end if
14.  upon RepSignal at  $R_j$  do
15.     $reply_{ij} \leftarrow shared\_mem.buffer\_toSend.read()$ ;
16.     $reply_{ij}.id \leftarrow id_{R_j}$ ;
17.    send  $reply_{ij}$  to  $C_i$ ;
18.     $replycache \leftarrow replycache + reply_{ij}$ ;
19.  upon  $(timer() - \delta_p.time) \geq period_{C_i}$  at  $R_j$  do
20.     $state_{ij} \leftarrow thread.Checkpoint()$ ;
21.     $hash_{ij} \leftarrow calculateHash(state_{ij})$ ;
22.     $shared\_mem.buffer\_Checkpoint_{R_j}.write(hash_{ij})$ ;
23.     $\delta_p.time \leftarrow t()$ ;

```

---

```

{ Agreement Service AS }
24. upon ReqSignal at AS do
25.   while  $\forall j \exists (req_j) \in shared\_mem.repBuffer_{R_j}$  do
26.     if  $req_j \notin orderedList$  then
27.        $shared\_mem.buffer\_toExecute.write(req_j)$ ;
28.        $orderedList \leftarrow orderedList \cup req_j$ ;
29.        $\delta_{int}.time \leftarrow timer()$ ;
30.        $\delta_{int}.id \leftarrow req_{i}.id_m$ ;
31.     else
32.       discard( $req_j$ );
33.     end while
34.  upon RepSignal at AS do
35.   while  $\forall j \exists (rep_j) \in shared\_mem.repBuffer_{R_j}$  do
36.      $buffer(rep_j) \leftarrow buffer(rep_j) \cup \{shared\_mem.repBuffer_{R_j}.read()\}$ ;
37.     if  $|buffer(rep_j)| \geq f+1$  then
38.       if match( $buffer(rep_j)$ ) =  $f+1$  then
39.          $Sign_{AS} \leftarrow Signature(\langle REPLY, rep_{ij}.id_{C_i}, rep_{ij}.resp \rangle)$ ;
40.          $reply_k \leftarrow \langle REPLY, null, id_{C_i}, rep_{ij}, Sign_{AS}, Cert_{AS} \rangle$ ;
41.          $shared\_mem.buffer\_toSend.write(reply_k)$ ;
42.          $\delta_{int}.time \leftarrow t$ ;
43.         restoreVMS( $f+1$ );
44.       else
45.         initiatedVMS( $f, bufferHash$ );
46.       end if
47.     end if
48.   end while
49.  upon  $(timer() - \delta_{int}.time) \geq timeout_{AS}$  at AS do
50.    initiatedVMS( $f, bufferHash$ );
51.     $\delta_{int}.time \leftarrow timer()$ ;
52.  upon CkPointSignal at AS do
53.   while  $\forall j \exists (hash_{ij}) \in shared\_mem.buffer\_Checkpoint_{R_j}$  do
54.      $bufferHash \leftarrow bufferHash \cup \{shared\_mem.buffer\_Checkpoint_{R_j}.read()\}$ ;
55.   end while

```

---

Após definida a ordem de seqüência de uma requisição, as réplicas são sinalizadas da presença da mesma no *buffer*  $buffer\_toExecute$  (linha 7). A réplica  $R_j$  faz a leitura da requisição ordenada (linha 8), e uma vez comprovada a sua autenticidade (verificação da assinatura linha 9), executa a operação  $op$  (linha 10) enviada pelo cliente. Na seqüência,  $R_j$  compõe com os resultados desta execução ( $resp_k$ ) a mensagem de resposta  $rep_k$  (linha 11). Esta mensagem é inserida na memória compartilhada, em outro *buffer* de escrita da réplica ( $shared\_mem.repBuffer_{R_j}$ ) na linha 12.

Com a escrita nos *buffers*  $repBuffer_{R_j}$ , o *Agreement Service* é ativado por sinalização correspondente (linha 34). As mensagens  $rep_k$  são então lidas e inseridas em *buffers* específicos para respostas de cada requisição  $buffer(rep_k)$  (linhas 35 e 36). Ou seja, todas as respostas recebidas das réplicas para uma requisição são inseridas em um mesmo *buffer* ( $buffer(rep_k)$ ).

Se o número de mensagens inseridas em um destes *buffers* (específicos para mensagens com os resultados de processamento de uma dada requisição de cliente) atinge o limite  $f+1$  e todas as mensagens apresentam os mesmos resultados de processamento (linhas 37 e 38), o *Agreement Service* cria então a sua assinatura sobre alguns campos e

monta uma mensagem de *REPLY* autenticada ( $reply_k$ ) para ser enviada ao cliente  $C_i$  (linhas 39 e 40). Na sequência, o *AS* torna a mensagem  $reply_k$  disponível no *buffer*  $buffer\_toSend$  (linha 41). As condições verificadas nos testes das linhas 37 e 38 representam que não houve desacordo sobre as respostas das  $f + 1$  réplicas de execução que processaram a requisição  $req_k$ .

Como consequência do recebimento pelo *AS* de pelo menos  $f + 1$  mensagens de resposta ( $reply_k$ 's à requisição  $req_k$ ) corretas e com resultados de processamento iguais, o temporizador que controla o  $timeout_k$  deve ser desativado (linha 42). E a função  $restoreVM(f+1)$  é chamada para restaurar a configuração mínima caso tenha havido a evolução da ME para  $2f + 1$  réplicas (linha 43).

A condição *else* da linha 44 corresponde a situação em que o teste envolvendo a função  $match()$  (linha 38) identificou desacordo nos  $f + 1$  resultados de processamento das réplicas da ME. Neste caso, o *AS* precisa ativar  $f$  novas réplicas para que se consiga o acordo necessário. Esta ativação se dá com o uso da função  $initiatedVMs()$  da linha 45, passando como parâmetros na mesma, o número de réplicas a serem ativadas (valor  $f$ ) e o resumo do estado correto salvo no último *checkpoint* realizado no sistema.

Se o temporizador associado com o prazo  $timeout_k$  expirar sem o recebimento de todas as respostas, o tratador descrito entre as linhas 49 e 51 é ativado. Nesta situação, a função  $initiatedVMs()$  é também chamada para criar  $f$  novas réplicas, tentando fazer com que o esquema chegue ao acordo nas respostas de processamento da requisição  $req_k$  (linha 50). Na sequência, na linha 51, o  $timeout$  interno é reativado, dando o prazo para que as novas réplicas atualizem seus estados e executem  $req_k$ .

A réplica  $R_j$  é ativada pela sinalização do  $RepSignal$  (linha 14), lendo então  $reply_k$  (resposta assinada para a requisição  $req_k$ ) no *buffer*  $buffer\_toSend$ . De posse da resposta assinada as réplicas inserem o seu identificador  $id_{R_j}$  nesta mensagem e enviam a mesma para o cliente (linhas 16 e 17). Além disso, uma cópia da resposta é armazenada em seu *cache* local para atender a possíveis retransmissões (linha 18). Bastando que uma destas mensagens de  $reply_k$  atinja o cliente, tenha a sua validade comprovada pela assinatura e certificado do *AS* que acompanham a mesma, o protocolo finaliza a sua execução.

Nas linhas entre 19 e 23 de uma réplica  $R_j$ , é descrito um procedimento de *checkpoint* que é executado periodicamente nas réplicas de execução. A função  $thread\_Checkpoint()$  é chamada, salvando o estado da aplicação executada e registros de controle usados em nível de réplica/*VM* (linha 20). Um resumo é calculado do estado da réplica

usando a função *calculateHash()* que usa no cálculo deste resumo uma função de *hash* criptográfico (linha 21). O resumo do estado da réplica (*hash<sub>ki</sub>*) é então passado ao *AS* via memória compartilhada (linha 22). Na sequência o código da réplica prepara o temporizador para o próximo *checkpoint* (linha 23).

O *AS* ao receber um sinal de *checkpoint* (*CkPointSignal*, linha 52), busca os resumos *hash<sub>ki</sub>* na memória compartilhada e os armazena em *bufferHash* (linha 53).

#### 4.2.4 Ativação de Novas Réplicas no AS

O algoritmo 3 ilustra a execução da função de ativação de novas réplicas no *AS*. Primeiro, é chamada a função *initiatedVMs()*, passado o número de réplicas necessário (*f*) para resolver o desacordo e o *buffer* de resumos de estado (*bufferHash*, linha 3).

Antes de ativar uma nova réplica o *Agreement Service* precisa passar como parâmetro o intervalo em que a nova réplica precisa executar para atualizar o seu estado. O valor inicial é obtido através do último *checkpoint* acrescido de 1 (linha 4) e o valor final refere-se ao número da requisição em que houve o desacordo (linha 5). Ou seja, esses dois valores serão usados pela nova réplica para obter o estado mais atual antes do desacordo. Após obter esses valores o *AS* escreve na memória ficando assim disponível o intervalo de execução do estado para a nova réplica. Enquanto, na linha 7 e 8 são iniciadas as *f* novas réplicas necessárias para resolver o desacordo.

---

Algoritmo 3. Protocolo de Ativação de Novas Réplicas no AS

---

**Variables**

1. *newReplicasList* = {*f*}; % conjunto de réplica pausadas
2. *updateRange* [ ] [ ] =  $\emptyset$ ; % requisições que precisam ser executadas para

**Init**

3. **function** *initiatedVMs*(*f*, *bufferHash*) % atualizar o estado da réplica
  4. *updateRange.first*  $\leftarrow$  |*LastCheckpoint\_Update#*| + 1; % primeira requisição depois do último C.P
  5. *updateRange.Last*  $\leftarrow$  *Disagreement\_Update#*; % requisição em que houve o desacordo
  6. *shared\_mem.buffer\_update.write*(*updateRange*); % escreve o intervalo na memória
  7. **For each** replica  $R_j \in$  *newReplicasList*
  8.     *StartVM*(*R<sub>j</sub>*); % inicia a réplica
  9. **end for**
- 

Essas novas réplicas buscam o estado da aplicação do último *checkpoint* nas réplicas de execução e verificam se o mesmo é referente aos valores de *hash* enviados pelo *Agreement Service*. Se o estado for válido recupera o mesmo, executa as requisições necessárias para chegar à requisição que apresentou a falha para resolver o desacordo gerado.

As réplicas novas quando retornam, começam pela sinalização *ReqSignal* da linha 7 do algoritmo 2, devido a presença de  $req_k$  em *shared\_mem.buffer\_toExecute* que ainda não foi processada por estas novas réplicas. Ao executarem a requisição  $req_k$ , enviam suas respostas (mensagens  $rep_k$ , linha 12 do algoritmo 2) ao *AS* que volta a executar os mesmos testes do algoritmo 2 na verificação se foi atingido o acordo (linhas 37-38 do algoritmo 2). A introdução de  $f$  novas réplicas aumenta o número de respostas para  $2f + 1$ , transpondo a impossibilidade de decisão sobre as respostas ocorrida com somente  $f + 1$  réplicas, e desta forma podendo ser identificada a resposta correta do conjunto (a resposta que apresentar maioria é considerada a resposta correta). A obtenção da resposta correta permite a continuidade da terminação do algoritmo.

O cliente neste modelo também tem papel ativo. Caso o cliente não receba uma resposta para sua requisição dentro de prazo definido ou as respostas recebidas são corrompidas, o mesmo retransmite a requisição. Os servidores que receberem a requisição assumem o comportamento normal do algoritmo 2, nas linhas 2 e 3, e inserem a requisição retransmitida em seus respectivos *buffers reqBuffer<sub>rj</sub>*. O serviço *AS* detecta então a retransmissão e ativa novas VMs incrementando a replicação para  $2f + 1$  réplicas. Uma vez ativadas, as novas tarefas vão se comportar como descrito na seção 4.2.3.

Uma vez concluído o processamento e a terminação do algoritmo, o número de  $2f + 1$  VMs (réplicas) é diminuído para  $f + 1$  com a identificação e a remoção das réplicas do servidor consideradas comprometidos (*restoreVMs* linha 41) e que levaram ao desacordo nas respostas na configuração inicial do servidor ( $f + 1$  réplicas iniciais).

Outra função que o método *restoreVMs* executa é o retorno do número de réplicas para  $f + 1$  no caso onde a quantidade de réplicas em execução é maior que este limite. Isso pode ocorrer, por exemplo, quando consideramos um valor de  $f$  igual ou superior a 2. Neste caso, qualquer uma das réplicas corretas será desligada para retornar ao número de  $f + 1$  réplicas.

#### 4.2.5 Checkpoint das Threads de Execução

Periodicamente as réplicas executam o *checkpoint* do estado da aplicação. Para isso, é necessário o decurso do prazo *period<sub>ck</sub>*, na linha 19 do Algoritmo 2. Quando ocorre o fim deste prazo, a operação de *checkpoint* é executada (linha 20) e um resumo do estado é gerado (*Hash<sub>kj</sub>*, linha 21). Esse valor de *hash* é enviado para a memória compartilhada através do *buffer shared\_mem.buffer\_Checkpoint* (linha 22). O *AS* aguarda  $f + 1$  *hashes* iguais para o *checkpoint* correspondente e

então salva esses valores em um conjunto, que será utilizado na atualização de novas réplicas (linhas 52 a 54).

#### 4.2.6 Considerações sobre a Proposta Algorítmica Apresentada

Na seção 4.2.3 foi apresentada através do algoritmo 2 a execução de uma instância de consenso que ordena a requisição  $req_k$ . Esta requisição é inicialmente executada em  $f+1$  réplicas e quando não ocorre qualquer tipo de desacordo ou falhas, são produzidas e comparadas  $f+1$  respostas destas réplicas. Como consequência um  $reply_k$  é produzido e enviado na linha 41 (algoritmo 2) para as réplicas que então, por sua vez enviam a mesma ao cliente. No entanto, quando se tem a participação de réplicas maliciosas na replicação BFT as inconsistências ou desacordos vão ocorrer e devem ser detectados pelo *Agreement Service*. A não verificação das condições dos testes das linhas 37 e 38 (algoritmo 2) ou o decurso do prazo (do  $timeout_k$ ) controlado na estrutura  $\delta_{in}$  (linha 49 do algoritmo 2) são os mecanismos de detecção do desacordo e do comportamento malicioso de réplicas do sistema. Como no máximo  $f$  destes comportamentos anômalos podem ocorrer e sempre podemos ativar  $f$  novas réplicas, então este acordo sobre a resposta da ME sempre vai ocorrer. Neste caso, o *safety* está assegurado no nosso esquema.

Para a terminação do algoritmo, é necessário que condições favoráveis ocorram nas comunicações entre cliente e servidores. Como assumimos o comportamento *parcialmente síncrono* tanto na rede como na memória compartilhada, as nossas execuções de protocolo devem atravessar períodos de sincronismo para garantir as chegadas das mensagens entre as partes envolvidas. Outro aspecto que contribui para a terminação dos algoritmos é que também assumimos comunicações confiáveis.

Sobre as retransmissões do cliente precisamos ainda esclarecer detalhes da abordagem. Se no modelo de sistema distribuído tivéssemos assumido o comportamento de *fair links* [Reynal, 2005], mensagens poderiam ser perdidas. Os clientes ficariam esperando por uma resposta válida para poderem evoluir. A temporização armada do cliente provoca possíveis recuperações com retransmissões das réplicas e mesmo assim as mensagens chegariam nas réplicas (ou no cliente, se for o caso). Em canais confiáveis que é a hipótese assumida no nosso modelo de BFT, as comunicações das réplicas sempre devem chegar e, dentro de um tempo finito, se as condições favoráveis de sincronismo ocorrem no sistema (premissa de sistema distribuído parcialmente síncrono). Portanto, com canais confiáveis, um cliente sempre retornará para sua aplicação com uma resposta em tempo finito, assegurando a terminação do acordo.

Em qualquer situação de desacordo em respostas ou *checkpoint* com a configuração inicial ( $f + 1$  réplicas), novas réplicas são ativadas. Porém, para poder responder ao cliente, estas novas réplicas ao serem iniciadas precisam obter o estado anterior à requisição cujas respostas não atingiram o quórum exigido. Para isso, a réplica ao ser iniciada verifica em memória compartilhada qual é o *checkpoint* estável ( $state_{k-1}$ , linha 20 do algoritmo 2) e seu *hash* correspondente, e os usa para se atualizar, ficando pronta para iniciar sua operação normal.

O serviço replicado inicia sempre com  $f + 1$  réplicas, e sempre em situações de desacordo (em respostas ou *checkpoint*) aumenta o número de réplicas para  $2f + 1$ . As réplicas que contribuíram para o desacordo podem ser identificadas pelo *AS* nas comparações de respostas ou de *hashes*. Estas réplicas em desacordo com a maioria são desativadas, visto que as mesmas poderão influenciar no desempenho do algoritmo e porque estão fazendo uso dos recursos físicos de forma inadequada. As réplicas que vão compor a nova configuração inicial estavam entre aquelas que produziram o valor mais frequente (de resposta ou de *checkpoint*).

Mesmo com canais confiáveis, réplicas poderiam se negar a enviar ao cliente suas respostas autenticadas pelo *Agreement Service* ( $reply_k$ ). A redundância da configuração inicial ( $f + 1$ ) já é suficiente para que pelo menos uma mensagem correta possa atingir o cliente em condições favoráveis de sincronismo. Os canais confiáveis garantem isto.

Visto que a memória compartilhada possui nos seus diversos *buffers* capacidade de armazenamento finito é necessário que se faça a remoção das informações antigas. A ocorrência de uma sinalização de *checkpoint* vindo de réplicas determina que todos os registros referentes ao processamento de requisições anteriores ao novo *checkpoint* sejam removidos, através da ativação, pelo *AS* de um coletor de lixo.

### 4.3 SERVIÇOS REPLICADOS

A replicação Máquina de Estados (ME) que permite a tolerância a intrusões, descrita nas seções anteriores, é neste item colocado na forma de um serviço. Uma infraestrutura de serviços é definida para incorporar os conceitos e algoritmos do *ITVM*. Esta infraestrutura de serviços faz uso da composição de serviços *web* para implementar o nosso modelo de réplicas ativas. A composição de réplicas é fundamentada no conceito de *Orquestração Descentralizada* [Chafle, et al., 2004] que mantém fluxos de execução evoluindo de forma descentralizada na coordenação de composições. A literatura tem sido profícua na discussão desta coordenação de composições por fluxos descentralizados. Em [Chafle, et al., 2004] são apresentados mecanismos para o particionamento do fluxo



de execução de uma composição em vários *engines* distribuídos. A motivação destes e de outros autores em suas propostas é diminuir a concentração de todo o fluxo de controle de uma aplicação distribuída sob uma entidade única (um coordenador geral que é caracterizado por um único *engine*), encontrada em modelos convencionais de orquestração.

Em nosso modelo, portanto, *engines* distribuídas definem os fluxos de controle e de dados, através dos serviços replicados, na concretização das trocas de mensagens entre o cliente e as aplicações servidoras. A Figura 4.5 ilustra uma composição de serviço *web* usando o nosso *framework*, onde as réplicas executam a mesma requisição de operação. Cada réplica de serviço está associada a um *engine* da orquestração distribuída, que executa a parte que lhe cabe do protocolo BFT de replicação ativa, que neste caso é o *ITVM*. Ou seja, os fluxos de execução do protocolo citado foram estruturados através da linguagem BPEL [Arkin e Askary, 2004], permitindo assim uma orquestração descentralizada da ME tolerante a intrusões. Os *engines* da Figura 4.5 que fazem parte do modelo, definem um modelo hierárquico de composições que interagem entre si apresentando a ideia de uma orquestração distribuída. O *engine* cliente pode ser interpretado como uma composição de um nível (hierárquico) superior em relação às composições que concretizam as réplicas do serviço. Esta infraestrutura de serviços que implementa o nosso modelo é realizada sobre um conjunto de máquinas virtuais e usa mecanismos de memória compartilhada para se comunicarem como enfatizado anteriormente neste texto.

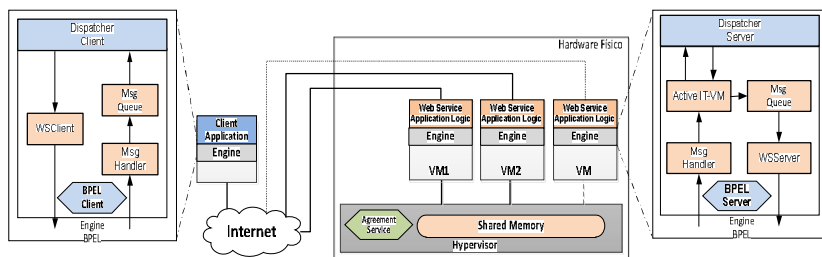


Figura 4.5 – Framework de serviços em replicação ME.

### 4.3.1 Componentes e suas Interações no Framework Desenvolvido

Cada *engine* do *framework* da figura contém os mecanismos necessários para gerenciar as interações do mesmo com o *engine* do cliente e com a memória compartilhada (através do ambiente virtualizado). De forma mais precisa, podemos dizer que um *engine* é

composto por uma unidade geradora de fluxo e de componentes que distribuem este fluxo através de interações com o ambiente em que o *engine* atua. As interações de componentes de um *engine* devem então atuar com o ambiente de virtualização (*Agreement Service* e memória compartilhada) e coordenar a réplica correspondente na execução das requisições e envio de respostas ao cliente. A Figura 4.5 ilustra alguns destes componentes. No lado do cliente, temos o *BPEL Client*, que é o componente responsável pela geração de parte do fluxo de controle que envolve a execução de uma requisição de um cliente no ambiente distribuído. É através deste *BPEL Client* que o *engine* no cliente ativa os componentes *Msg Handler*, *Msg Queue*, *WSClient*, *Dispatcher Client*, fazendo que os mesmos atuem junto ao ambiente do cliente. Uma descrição mais específica destes componentes é apresentada na Tabela 4.1.

Tabela 4.1 - Componentes internos dos engines.

<b><i>BPEL Client</i></b>	É responsável pela execução do fluxo de controle para a execução remota de requisições do cliente. As sinalizações deste componente para outros componentes internos no cliente envolvem a montagem das requisições, a assinatura das mesmas e a recepção de respostas e suas verificações.
<b><i>Dispatcher Client</i></b>	Componente que fornece uma interface entre a aplicação cliente e o <i>engine</i> . Tem a função de encapsular informações para realizar as invocações nas réplicas e obter do <i>Msg Queue</i> as respostas a serem entregues para a aplicação.
<b><i>WSClient</i></b>	Componente responsável em obter informações do serviço remoto (referência do serviço web, o método a ser executado e os parâmetros necessários para a sua invocação nas réplicas). Este componente é também responsável pela invocação transparente das múltiplas instância de serviços web com quem o cliente interage. Estas interações fazem uso do protocolo <i>WS-ReliableMessaging</i> .
<b><i>BPEL Server</i></b>	É responsável pela execução do fluxo de controle através da sinalização a outros componentes do <i>engine</i> de réplica de serviço na comunicação com os clientes. Além disso, gerência as trocas das réplicas, com os mecanismos de memória compartilhada definidos pelo protocolo ITVM.
<b><i>WSServer</i></b>	Este componente é responsável em obter uma resposta assinada no <i>Msg Queue</i> e preparar a resposta correspondente a ser enviada ao cliente.
<b><i>Active ITVM</i></b>	Este componente atua na execução do protocolo ITVM juntamente na comunicação com a área de memória compartilhada. Após receber uma notificação do <i>Msg Handler</i> executa as operações que envolvem a ordenação (sinaliza uma notificação para o <i>Dispatcher Server</i> ), inserção da resposta (executada após receber uma resposta do <i>Dispatcher Server</i> ) e a validação e a assinatura da resposta. A resposta assinada pelo <i>AS</i> é transferida para o <i>Msg Queue</i> e uma notificação é enviada ao <i>WSServer</i> aguardando para ser processada.
<b><i>Msg Queue</i></b>	<i>Buffers</i> onde requisições são armazenadas antes de serem consumidas pelas réplicas correspondentes ou para armazenamento das respostas. Estes <i>buffers</i> são também usados nas eventuais retransmissões de mensagens.
<b><i>Msg Handler</i></b>	A função destes tratadores envolve a geração de mensagens de reconhecimento e pedidos de retransmissão. No lado do cliente, quando uma mensagem é recebida, uma notificação é enviada para <i>BPEL Client</i> e o componente <i>Dispatcher Client</i> , sendo então a mesma transferida para o <i>Msg Queue</i> onde deve aguardar para ser processada pela aplicação. No lado servidor, o tratador sinaliza a chegada de requisição para o <i>BPEL Server</i> e a mesma é enviada para o <i>Active ITVM</i> .

No lado servidor, temos os *engines* das réplicas que também são formadas por um componente principal identificado como *BPEL Server*. Este componente é o gerador de fluxo que ativa os blocos de interação do

*engine* de réplica. A evolução deste fluxo define então as invocações do serviço de aplicação e as interações relacionadas com a execução do protocolo *ITVM*. Os componentes de interação deste *engine* de réplica devem, portanto, envolver interações implementando as comunicações via memória compartilhada, as chamadas de operações na réplica de execução e ainda a concretização do envio de respostas. Os componentes que fazem parte das *engines* de réplicas são: *Active ITVM*, *Msg Queue*, *WSServer*, *Msg Handler*, *Dispatcher Server*. Os mesmos também estão descritos na Tabela 4.1.

Os fluxos de execução definidos a partir dos componentes *BPEL Client* e *BPEL Server* têm, portanto, a função de interagir com os componentes internos dos *engines* cliente e de réplica, respectivamente. É através destes fluxos que os demais componentes internos de *engines* são chamados para exercerem suas funções. Na montagem de uma requisição, *engine* cliente (*BPEL Client*) interage com os componentes *Dispatcher Client* (fornece uma interface entre *engine* e o cliente) e *WSClient* para assinar a requisição e encapsular os parâmetros para execução da operação e o correspondente envio. Enquanto, no lado do servidor, o fluxo do *engine* (*BPEL Server*) da réplica irá interagir com os componentes para o recebimento da requisição que passa pelo componente *Msg Handler* e encaminha a requisição para o *Active ITVM* (que executa as operações relacionadas ao protocolo *ITVM* para operações na parte de memória compartilhada). Após a ordenação via *Active ITVM* a requisição é encaminhada para o componente *Dispatcher Server* (interface entre o *engine* e a réplica) para ser executada pela aplicação. Ao final, uma resposta é novamente encaminhada para o *Active ITVM* que interage com o *AS* para a assinatura da resposta e a posterior transferência da mesma para o *Msg Queue*. Uma sinalização será enviada para o *WSServer* e o *Dispatcher Server* informando que uma resposta assinada se encontra para ser consumida. O *WSServer* busca a resposta assinada e envia para o cliente.

Neste modelo, para garantir a correção da execução do algoritmo distribuído, precisamos da premissa de que as interações entre cliente e réplicas se dão através de comunicações ponto a ponto confiável. Para construir a correção das comunicações ponto a ponto fazemos uso da especificação *WS-ReliableMessaging* (WS-RM) [WS-RM, 2004]. A implementação das recomendações desta especificação envolve componentes *engines* do *framework* (*WSClient*, *Msg Handler* e *WSServer*, respectivamente).

Para explicitar o funcionamento, considere ainda o código de aplicação cliente enviando uma requisição. Este mesmo cliente deve

bloquear esperando por uma resposta válida dos servidores, uma vez que, o cliente espera por uma resposta assinada pelo *Agreement Service*. Se o cliente não recebe qualquer informação durante certo intervalo de tempo, a requisição é reenviada. Neste caso, o componente *BPEL Client* chama o componente *WSClient* para o reenvio da requisição. Caso a requisição já tenha sido processada, os servidores apenas reenviam uma cópia de suas respostas disponíveis na memória compartilhada.

#### 4.4 PROTÓTIPO E TESTES

Um protótipo de serviço replicado foi implementado usando a tecnologia de *Web Services* visando testar as nossas proposições. Neste protótipo cliente e servidor se executam em uma rede Ethernet de 100 Mbps conectado por uma máquina servidora (máquina física) com processador *Core I7*, 8 GB de RAM.

A camada de virtualização do nosso modelo utiliza o *hypervisor* XEN [XEN, 2009], o qual, além de executar e gerenciar as máquinas virtuais, também oferece um recurso para a implantação da memória compartilhada, denominado *XenStore*. Este recurso é responsável pelos controles de acesso e de concorrência sobre essa memória compartilhada. O *hypervisor* usado oferece ainda um domínio especial, denominado *Domain0*, que foi utilizado na nossa implementação como domínio confiável para a execução do *Agreement Service* que é um componente essencial do nosso modelo. Sobre o *hypervisor* são então executadas as máquinas virtuais (réplicas) que processam as requisições de serviço. Estas máquinas virtuais executam diferentes versões do Sistema Operacional Linux atendendo os requisitos de diversidade necessários para se conseguir a independência das vulnerabilidades das réplicas.

O *framework* desenvolvido foi implementado fazendo uso do *Apache ODE* [Apache 2009]. Os softwares utilizados do *Apache ODE* permitiram a criação das orquestrações necessárias para execução do nosso modelo. A modelagem dos fluxos de execução foi descrita através da linguagem BPEL. O cliente neste protótipo enxerga a replicação ME implementada sobre um conjunto de máquinas virtuais como um Serviço *Web*. Toda a estrutura de *engines* que o cliente necessita e também informações de acesso a este serviço podem ficar disponíveis a partir de um UDDI como usualmente se faz para qualquer composição de serviços. O restante da aplicação foi desenvolvido em *Java*.

Uma aplicação bancária foi implementada neste protótipo onde operações de crédito e de débito são recebidas pelo serviço replicado em favor do cliente. As requisições destas operações e suas respostas são assinadas e validadas usando criptografia assimétrica RSA de 1024 bits.

Visando verificar o desempenho da infraestrutura de serviços e do modelo de tolerância a intrusões implementado na mesma, testes foram executados sobre o protótipo desenvolvido. Nos vários testes realizados, foi considerado o limite para réplicas maliciosas como  $f = 1$ . Este valor de  $f$  foi escolhido pela falta de recursos da máquina física hospedeira para manter muitas máquinas virtuais. Os testes realizados foram executados em dois cenários distintos: (i) ambiente sem réplicas maliciosas, caso otimista, e (ii), ambiente com réplicas maliciosas. No cenário com réplicas maliciosas 20% das requisições enviadas envolvem execução de réplicas maliciosas e, portanto, apresentam desacordo, ou seja, a cada 5 requisições o processo de recuperação de uma nova réplica foi executado.

A Figura 4.6 demonstra o tempo médio de resposta no cliente nos dois cenários. Enquanto no cenário sem falhas o tempo de resposta foi 28,32 ms no cenário com falhas esse tempo médio de resposta passou a ser de 245,49 ms. Esse aumento se deve ao fato de que a cada requisição onde ocorre o desacordo das respostas,  $f$  novas réplicas devem ser ativadas para resolver essa falha. Além da ativação dessas  $f$  novas réplicas existem ainda o processamento da identificação da réplica maliciosa e a remoção da mesma. Todo esse processo de recuperação do protocolo tem tempo médio de 1100 ms. Ainda que o tempo médio de resposta no cenário com falhas tenha um valor extremamente maior que no cenário otimista, o sistema continuou respondendo corretamente para o cliente mesmo com uma grande presença de falhas.

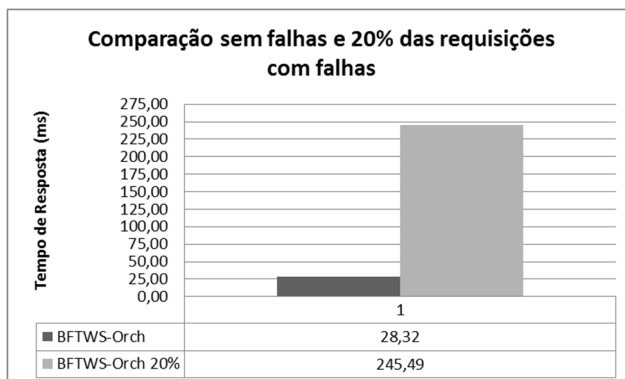


Figura 4.6 – Comparação da arquitetura com falhas e sem falhas.

A Figura 4.7 ilustra uma comparação do tempo de resposta considerando a utilização por usuários simultâneos. Neste cenário comparamos o nosso modelo de BFT com a abordagem SMIT que é muito

similar ao nosso por usar máquinas virtuais. A abordagem SMIT é descrita na seção de trabalhos relacionados. Neste cenário da Figura 4.7, comparamos os tempos médios de resposta dos casos com falhas e sem falhas de ambas abordagens. O tempo de resposta é impactado em todos os testes deste cenário, visto que com o aumento do número de clientes, a utilização e concorrência sobre os recursos da máquina física são incrementadas. A nossa abordagem tem um melhor desempenho em termos de tempo de resposta. Isso se deve ao fato de envolver configurações com menos réplicas ( $f + 1$ ) e, portanto, menos VMs. Com o número de réplicas reduzido, a possibilidade de uma melhor utilização dos recursos do servidor físico melhora o desempenho da replicação. Outro ponto em que a nossa abordagem apresenta vantagens é na ocorrência de falhas maliciosas. Enquanto o SMIT, na ocorrência de falhas, as réplicas comprometidas continuam na execução do protocolo, influenciando no desempenho do mesmo, em nosso modelo as réplicas maliciosas são continuamente removidas, de forma que o protocolo sempre seja reconfigurado para ser executado com somente  $f + 1$  réplicas corretas.

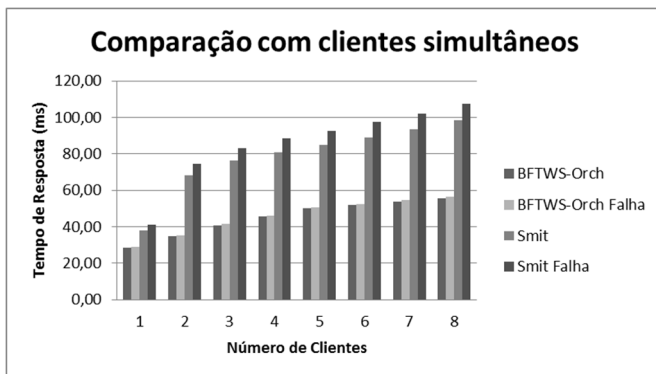


Figura 4.7 – Tempo de resposta considerando usuários simultâneos.

## 4.5 CONCLUSÃO

Neste capítulo foi introduzido um modelo híbrido que trata com diferentes tipos de faltas. O uso da virtualização e a definição e isolamento de elemento confiável nos leva a um conjunto de simplificações. Neste modelo existe a separação de faltas maliciosas e faltas de *crash*. Na verdade neste capítulo, só tratamos intrusões (faltas maliciosas ou bizantinas). A execução de um protocolo de consenso BFT fica muito

simplificado quando executado sobre memória compartilhada. O modelo também enfatiza a separação do acordo e da execução da ME. O custo do protocolo na situação otimista é de 2 passos de comunicação isto porque a maioria das etapas do acordo BFT são internos, executados em memória compartilhada. A proposta foi desenvolvida para WANs e considerando intrusões e falhas maliciosas. O número de réplicas varia entre  $f+1$  (caso otimista) e  $2f+1$  (em presença de falhas). Se levarmos estes dados para a Tabela 2.2 que compara propostas no capítulo 2, veremos que a nossa proposta apresenta vantagens em relação a muitas das propostas da literatura.





## 5 SERVIÇO TOLERANTE A INTRUSÕES EM REDES DE LONGA DISTÂNCIA

No capítulo anterior, apresentamos um modelo que separa faltas maliciosas de *crashes* em máquinas físicas. A tolerância a intrusões é conseguida com a implementação de replicação Máquina de Estados sobre máquinas virtuais que executam em um mesmo servidor físico (uma mesma máquina física).

Embora haja os avanços descritos no capítulo anterior, as soluções apresentadas até este ponto ainda estão longe dos objetivos que foram colocados nesta tese. Queremos serviços tolerantes a intrusões em redes de longa distância (*WANS*) como a Internet. Apesar da redundância de uma replicação ME, muitas situações de ataques e intrusões não são resolvidas devido à concentração desta replicação na mesma máquina física. No entanto o serviço replicado resolve o problema como ataques de *DoS*<sup>8</sup> com replicação física.

A solução que procuramos deve ter também um sentido prático, factível no sentido da implementação e que não roube muita banda passante dos servidores nas sincronizações necessárias da ME. E que não envolva latências muito grandes que afastem seus usuários nestes ambientes de longa distância. São estas as preocupações que guiaram os trabalhos apresentados no presente capítulo. Começamos com o modelo de sistema que já sofre umas pequenas modificações.

### 5.1 MODELO DE SISTEMA

O modelo apresentado nesta seção será o mesmo apresentado no capítulo anterior. O suporte de replicação Máquina de Estados (ME) continua assumindo um conjunto  $C = \{C_0, C_1, \dots, C_{n-1}\}$ , representando um número finito de processos clientes que enviam requisições para os servidores através da rede. Além do conjunto de clientes, o modelo mantém o conjunto de servidores  $S = \{S_0, S_1, \dots, S_{n-1}\}$  que concretizam a replicação do serviço. Só que aqui os servidores que continuam executando em VMs são particionados em sítios, caracterizando uma estrutura hierárquica de MEs.

---

<sup>8</sup> *Denial of Service* (ou negação de serviço) que é a forma mais usual de violação onde ataques visam tornar um serviço indisponível aos seus usuários. Normalmente, as soluções para estas formas de ataque envolvem redundância física de um *cluster*, em rede local, atendendo no endereço do serviço na Internet.

No modelo, continuamos separando processos (clientes e servidores) em corretos ou faltosos. Processos são ditos corretos quando seguem suas especificações propostas no suporte algorítmico. Enquanto, processos faltosos (ou maliciosos) se comportam de maneira arbitrária, podendo omitir mensagens, modificar o conteúdo das mensagens ou apresentar qualquer tipo de comportamento não especificado. Este tipo de comportamento malicioso também é chamado de bizantino ou intrusivo. Processos clientes possuem identificadores únicos garantidos por seus certificados e, por assinarem suas mensagens, não conseguem deixar os servidores replicados inconsistentes.

As réplicas de uma ME (réplicas servidoras), executadas em um sítio, são mantidas isoladas em máquinas virtuais próprias. Este isolamento permite a alocação de uma ou mais réplicas na mesma máquina física sem que as mesmas possam causar danos entre si. A essência do modelo de uma ME em um sítio continua a mesma em relação ao capítulo anterior. Ou seja, separamos também as réplicas da ME que executam as requisições do cliente, do mecanismo de acordo local executado através de um *Agreement Service* (AS), em cada sítio. Além disso, a memória local é o meio usado para a comunicação entre réplicas e o *Agreement Service*.

Além disso, no modelo assumimos também um comportamento parcialmente síncrono [Dwork, et al., 1988]: o sistema opera de modo assíncrono em boa parte do tempo, mas ocorrem períodos estáveis nos quais comunicações e processamentos são concluídos dentro de prazos finitos de tempo. Este modo de sincronismo garante a terminação dos algoritmos desenvolvidos. No modelo, os canais de comunicação são neste capítulo assumidos como *fair links*<sup>9</sup>. Isto implica que podem ocorrer perda de mensagens, mas estas não são significativas em relação as mensagens que chegam aos seus destinos.

Também assumimos como premissa a propriedade *AWBI* introduzida em [Fernandez, et al., 2007]. Isto é, um processo não deve executar indefinidamente de forma assíncrona. Como consequência, a memória compartilhada por processos sempre permitirá acessos em

---

<sup>9</sup> Canais de comunicação são ditos *fair links*, ou seja, mensagens podem ser perdidas nestes canais. Mas, se o emissor envia (ou retransmite) mensagens um número infinito de vezes e o receptor neste canal recebe um número infinito de mensagens deste mesmo emissor [Raynal, 2005].

tempo limitado, mas não necessariamente conhecido. Assumimos com isto que a memória compartilhada do modelo proposto, também possui um comportamento *parcialmente síncrono*.

### 5.1.1 Abordagem de Replicação do Modelo

Em nosso modelo, propomos uma abordagem hierárquica para a tolerância a intrusões e faltas em ambientes de larga distância, a exemplo de outras experiências da literatura como [Kirsch e Amir, 2008] [Veronese, *et al.*, 2010]. Neste sentido, para diminuir latências e custos na WAN e a sustentação de replicação Máquina de Estados (ME) nestes ambientes, definimos em nosso modelo dois níveis de protocolos de consenso: o local e o global. No modelo, definimos a abstração *sítio* que corresponde a uma composição de réplicas (ou máquinas virtuais) alocada em uma máquina física. Cada sítio com o seu domínio caracteriza o nível local da proposta hierárquica. Uma réplica *coordenadora* é definida em cada replicação de sítio, cujo papel é intermediar as ações locais com os outros sítios do sistema global. No nível global (ou de coordenação), são então participantes somente os coordenadores de sítios. É definido também o papel de coordenação geral do sistema que é desempenhado pelo que chamamos de sítio líder ( $S_l$ ). Na verdade este papel de liderança geral é desempenhado pelo sítio líder através de seu coordenador local. A Figura 5.1 ilustra os componentes descritos acima.

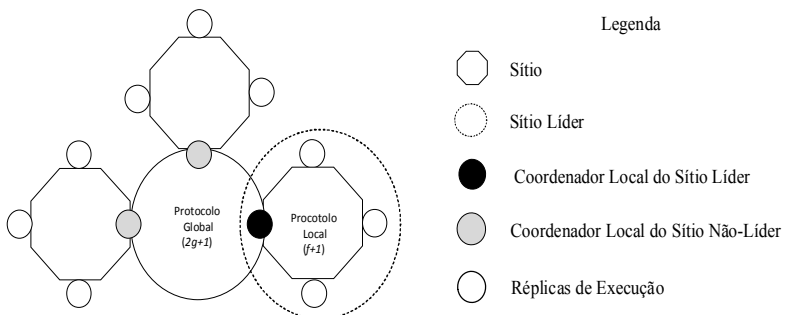


Figura 5.1 – Visão geral do modelo.

A abordagem hierárquica do modelo forma um esquema híbrido de tolerância a faltas que separa os diferentes tipos de falhas. Portanto, os diferentes níveis do sistema convivem com diferentes tipos de falhas. Em cada sítio (nível local do modelo), nós já vimos no capítulo anterior que a nossa proposta convive com faltas maliciosas ou bizantinas e que o número necessário de réplicas para a execução ME é de  $f + 1$  réplicas, no

caso sem ocorrência de malícia e de  $2f + 1$  quando ocorrem falhas maliciosas. O algoritmo de tolerância a faltas bizantinas (*BFT*) é um consenso executado sobre memória compartilhada muito simplificado porque envolve uma entidade confiável (*Agreement Service*).

Devido à separação de falhas, em nível global tratamos as possíveis falhas de *crash* das máquinas físicas onde estão alocados os sítios. Para isto usamos um algoritmo para o consenso entre os diferentes sítios do sistema, baseado em *rounds*, que é o *Paxos* [Lamport, 1998] [Lamport, 2001]. Ou seja, consideramos que a nível global o sistema evolui em presença de falhas de *crash*. Com isto, para a evolução do sistema, o número de sítios necessários é de  $2g + 1$ . Mesmo se considerarmos que coordenadores maliciosos participam a nível global, a ação maliciosa destes fica limitada ao não envio de mensagens que é facilmente modelada em *crashes*.

Para a comunicação entre sítios (a nível global), consideramos a premissa de *fair links* que é a assumida no algoritmo *Paxos* [Boichat, *et al.*, 2003]. Esta premissa de *fair link* é também assumida nas interações entre cliente e a replicação de servidores.

### 5.1.2 Componentes em Nível de Sítio

Em nossa proposta, fazemos uso da virtualização, de modo que, cada sítio seja formado a partir de máquinas virtuais (*VMs*) mapeadas em uma máquina física. A Figura 5.2 ilustra a organização interna de um sítio em nosso modelo.

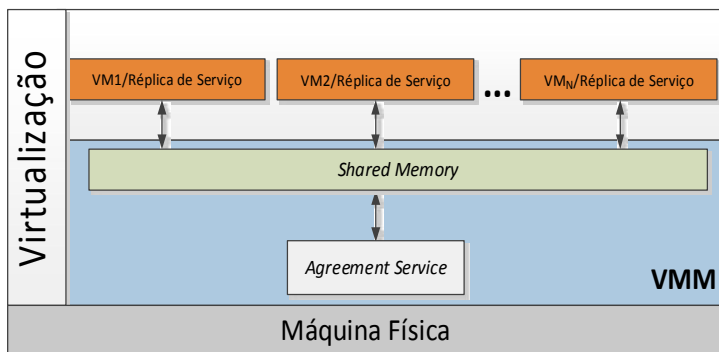


Figura 5.2 – Organização interna de um sítio.

O nível superior da figura citada apresenta as máquinas virtuais isoladas, executando suas respectivas réplicas de serviço. Estas *VMs* possuem diferentes portas e endereços de IP e o coordenador local está

sempre disponível aos clientes via *Internet*. Diante disto, como descrito anteriormente, assumimos neste nível faltas arbitrárias onde réplicas de serviços podem ser comprometidas por intrusões.

No nível abaixo das máquinas virtuais, está o Monitor de Máquinas Virtuais (*VMM – Virtual Machine Monitor*) que é o responsável pela execução da camada de virtualização do modelo. Este monitor tem por objetivo iniciar ou desligar as máquinas virtuais de acordo com a execução dos algoritmos locais de suporte da replicação ME. Nenhuma porta ou endereço deixa o VMM disponível externamente na rede. Com isto, assumimos que o *VMM* pode apresentar vulnerabilidades, porém estas não podem ser exploradas via rede ou mesmo pelas máquinas virtuais locais. Para que essa premissa seja garantida, confiamos que um *VMM* forneça o isolamento necessário para a execução segura de suas máquinas virtuais. Esta mesma premissa é usada em [Chun, *et al.*, 2008] [Sahoo, *et al.*, 2010].

Na camada do *VMM* são mantidas isoladas duas abstrações fundamentais ao modelo: a *Shared Memory* e o *Agreement Service (AS)*. O componente *AS* (Figura 5.2) é responsável pela ordenação das requisições de clientes, verificação das respostas das réplicas de serviço e ainda, pelas assinaturas de mensagens que são trocadas entre os sítios e das respostas que são enviadas para os clientes. Além disso, este serviço coordena a validação dos *checkpoints* feitos pelas réplicas de serviço e também, em conjunto com o *VMM*, mantém a dinâmica de ativação de novas réplicas em caso de desacordo nas respostas e em *checkpoints* das réplicas de execução. Ou seja, o *Agreement Service* trata dos aspectos não funcionais na implementação da replicação ME, de forma que este pode ser usado em qualquer tipo de aplicação sem modificações.

A separação indicada acima entre o gerenciamento das réplicas e a execução das mesmas, é também descrita em [Yin, *et al.*, 2003]. Desta forma, as réplicas de execução que executam a aplicação, ficam expostas aos clientes via rede, enquanto o *Agreement Service* que trata dos aspectos de gerenciamento da replicação, fica isolado e protegido no *VMM*.

O mecanismo de memória compartilhada (*Shared Memory*, na Figura 5.3), também mantido no *VMM*, oferece registradores para a comunicação entre as máquinas virtuais (réplicas executando a aplicação) e o *Agreement Service*. Basicamente são definidas duas operações sobre a memória: escrita e leitura. Cada máquina virtual do sítio contém seus *buffers* de escrita e leitura, assim como o *Agreement Service*. Uma política de controle de acesso é definida para que os *buffers* mantenham consistência e segurança nas operações de escrita e leitura entre estes

diversos componentes. Na Figura 5.3, a memória compartilhada apresenta a seguinte estrutura de *buffers*:

- *Buffers* individuais de cada *VM* usados para enviarem, ao *AS* do sítio, mensagens de protocolo recebidas por suas réplicas (*msgBuffer*) e também *buffers* individuais para passarem dados da aplicação de suas réplicas para o *AS* (*appBuffer*). Só as *VMs* proprietárias destes *buffers* podem escrever mensagens nos mesmos.
- *Buffers* onde só o *AS* é escritor que são o *buffer* para envio de requisições para a execução em réplicas (*buffer\_toExecute*) e o *buffer* para mandar só mensagens de protocolo para as réplicas do sítio (*protBuffer*).

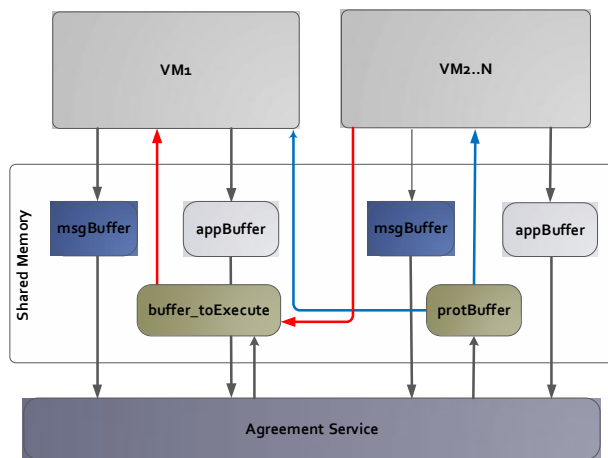


Figura 5.3 – *Buffers* na memória compartilhada.

O *Buffer* *buffer\_toExecute* conserva as mensagens enviadas para execução até as operações de checkpoint quando são removidas aquelas que se tornaram desnecessárias devido ao salvamento de estado.

A premissa básica para a correção dos sítios é então de que o *VMM* e as suas abstrações (a *Shared Memory* e o *AS*) sejam mantidos isolados da rede externa de forma que intrusões não possam ocorrer e alterar suas especificações.

### 5.1.3 Relação de Componentes em Nível Global

A utilização de replicação máquina estado implica em manter um grande número de servidores executando o mesmo serviço. Apesar destes mecanismos aumentarem a segurança no sistema, para o cliente todo acesso deve ocorrer de forma transparente. O usuário deve comunicar-se

somente com um servidor em seu acesso ao serviço e o suporte de replicação deve fazer o seu papel de manter o algoritmo da replicação máquina de estado em funcionando. Porém, no decorrer da execução do sistema, servidores que executam a aplicação acessada pelo usuário podem ser desligados ou removidos da ME. Nestes casos, o usuário deve desviar o envio de suas requisições quando não consegue acessar um servidor. Diante disto, ele envia para outro servidor que esteja disponível.

No entanto, diante desta necessidade, o usuário para o acesso ao serviço deve receber uma lista de servidores que podem ser usados na mediação com a ME. No nosso sistema consideramos esta lista como um *array* ( $coord_h$ ) com os identificadores e endereços de todos os coordenadores de sítios do sistema. Esta lista  $coord_h$  é a visão que um sítio  $S_h$  tem dos coordenadores dos sítios no sistema. E é claro que as várias listas mantidas por cada sítio podem apresentar inconsistências devido ao dinamismo do nosso sistema.

Os clientes devem acessar estas listas em repositórios de nomes, disponíveis nos diferentes sítios do sistema. Cada sítio usa o repositório de seu domínio de nomes para o registro desta sua lista de coordenadores. Esta maneira distribuída e independente de registrar suas listas em repositórios diferentes aumenta a disponibilidade do serviço replicado. A Figura 5.4 ilustra o registro de suas listas em seus repositórios de domínio de nomes por sítios da ME da nossa proposta. Depois deste registro, as informações destas listas ficam disponíveis para o acesso de usuário. Na figura, um usuário acessa uma destas listas através do uso do nome do serviço replicado.

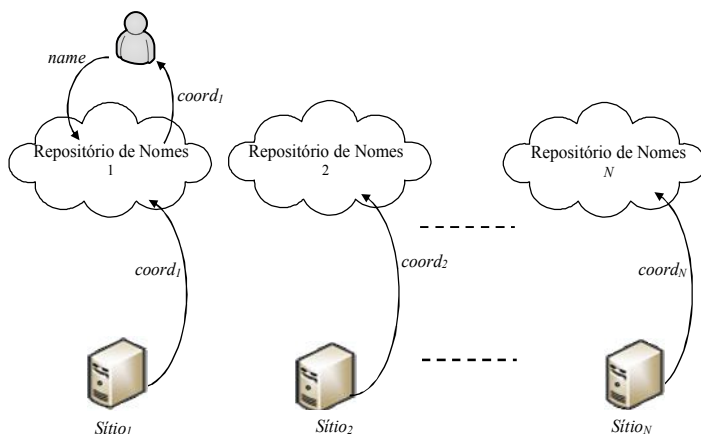


Figura 5.4 - Sítios e seus domínios de nomes.

Caso uma réplica coordenadora não responda, o cliente envia suas requisições para uma outra coordenadora dessa lista, em outro sítio do sistema. Com o objetivo de se manter esta lista de servidores acessíveis para o cliente, fizemos o uso do XRDS (*eXtensible Resource Descriptor Sequence*), um formato de arquivo XML para especificação e descoberta de metadados de acesso a provedores de serviço, utilizados em aplicações web. Neste arquivo são descritos os identificadores das réplicas coordenadoras juntamente com seus endereços de acesso de modo que estas possam ficar acessíveis via rede para atender a requisições do serviço de um cliente.

O cliente, no acesso ao serviço, busca em qualquer servidor dos sítios do protocolo (servidor WWW, servidores DNS, etc.) um arquivo XML no formato XRDS com os descritores dos servidores que atendem pelo serviço do cliente. Por exemplo, caso o domínio da aplicação do cliente seja “servico.com.br”, o cliente busca o arquivo na URL “www.servico.com.br/servidores.xml” e obtém a lista de servidores que podem atender pela aplicação do cliente, juntamente com uma ordem definida. Caso um servidor desta lista não responda às requisições do cliente, este envia suas requisições para os próximos servidores da lista.

## 5.2 VISÃO ALGORÍTMICA DO MODELO

Protocolos de consenso em ambientes maliciosos, mesmo em execuções favoráveis (sem a ocorrência de falhas maliciosas) assumem valores quadráticos no custo assintótico destes protocolos. Ou seja, protocolos que apresentam  $O(N^2)$  mensagens trocadas em cada consenso, sendo  $N$  o número de réplicas participantes cujo valor deve satisfazer a condição  $N \geq 3f + 1$ , e  $f$  corresponde ao limite de faltas ou intrusões admitidas no sistema.

No nosso modelo, em nível de sítio, também executamos algoritmo de consenso tolerante a faltas bizantinas. Nas condições de isolamento do *Agreement Service*, assumimos o mesmo como componente confiável no modelo. Devido ao uso dos elementos confiáveis na execução deste algoritmo, o número de réplicas necessárias para o consenso na presença de *VMs* corrompidas (maliciosas) cai para  $2f + 1$ . Em [Neves, *et al.*, 2004] são discutidos protocolos de BFT que fazem uso de elementos confiáveis (ou *wormholes*) e que apresentam suas necessidades de réplicas para o consenso diminuída para  $2f + 1$ .

Uma das vantagens no uso da virtualização em nossa proposta é o gerenciamento dinâmico das réplicas do sítio. Na execução de uma requisição de cliente, por exemplo, podemos tratar réplicas comprometidas (maliciosas). Inicialmente, um sítio começa a execução



de uma requisição de cliente com  $f + 1$  réplicas e somente no desacordo das  $f + 1$  réplicas em suas respostas é que novas  $f$  réplicas são ativadas, perfazendo as  $2f + 1$  réplicas necessárias para o consenso em presença de no máximo  $f$  réplicas maliciosas. No final, com o acordo envolvendo uma maioria de réplicas corretas, a resposta é enviada ao cliente e as réplicas que divergiram do valor acordado de resposta são desativadas, mantendo com isto o número de  $f + 1$  réplicas iniciais para a próxima execução de requisição. Com este procedimento o limite de réplicas maliciosas no sistema fica ilimitado, não se resumindo mais a somente  $f$  falhas de réplicas durante o ciclo de vida.

Como tratamos com faltas diferentes a nível global (em nível do *Paxos*) é necessário que o limite de faltas considerado seja também diferente. Ou seja, assumimos um limite  $g$  para as faltas de *crash* em nível global. Para que o consenso do *Paxos* ocorra, é necessário que tenhamos  $n$  sítios em nível global sendo  $n \geq 2g + 1$  (limite imposto para a resolução de consenso em ambiente de faltas de *crash*).

### 5.2.1 Descrição das Ações em Evolução Normal do Protocolo

No *Paxos*, quando o líder já é definido no início de uma instância de protocolo a fase de *prepare* se torna irrelevante. Esta versão do *Paxos*, com o líder já definido, é muito usada pelo seu melhor desempenho que envolve somente três passos [Kirsch e Amir, 2008] [Amir, *et al.*, 2010]. A Figura 5.5 ilustra esta situação. No nosso esquema também usamos esta versão com líder já predefinido.

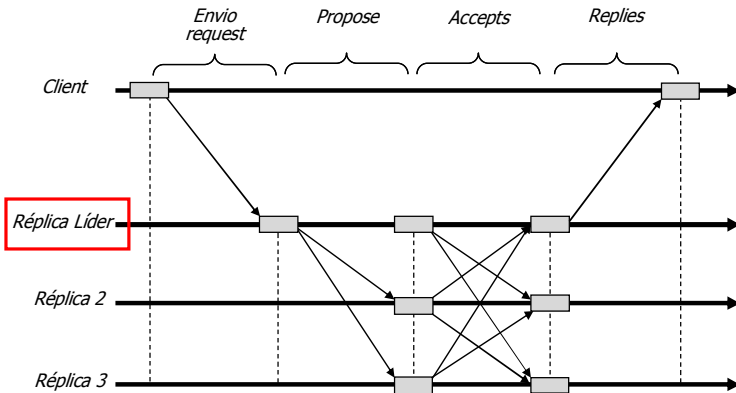


Figura 5.5 – *Paxos* com líder pré-definido.

Na Figura 5.6 apresentamos o nosso esquema executando em nível global<sup>10</sup>. No nosso protocolo global considerando que já saímos com sítio líder definido na *round* inicial de cada instância de protocolo e portanto temos uma versão de menos passos do *Paxos*.

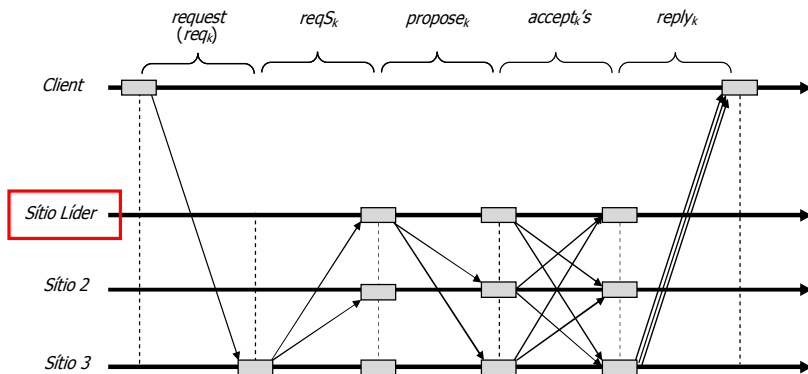


Figura 5.6 –Protocolo *Paxos* executado em nossa proposta.

Nesta Figura 5.6 é incluída um passo a mais em relação a Figura 5.5. Na verdade, este passo deveria ser indicado na versão simplificada do *Paxos* (da Figura 5.5), porque nem sempre o cliente conhece e envia a sua requisição ao líder de uma instância de protocolo. Mesmo no *Paxos* trocas de visão ocorrem, levando a situações em que o cliente deixa de ter acesso ao líder da ME [Kirsch e Amir, 2008].

No nosso esquema, o cliente fazendo uso da sua lista de coordenadores (*coord<sub>h</sub>*), seleciona uma réplica (de um sítio) e envia no seu endereço uma mensagem de requisição (*req<sub>k</sub>*). O coordenador do sítio, ao receber a mensagem do cliente, repassa a mesma ao seu *AS* para as verificações de assinatura e correção da mensagem recebida. Depois disto, o *AS* cria uma mensagem *reqS<sub>k</sub>* que inclui a requisição *req<sub>k</sub>* do cliente. Esta mensagem *reqS<sub>k</sub>* é também assinada, mas desta vez, pelo *AS* do sítio que recebeu a mensagem do cliente. Esta mensagem assinada (*reqS<sub>k</sub>*) é uma prova da admissão da mensagem no sistema. Depois, desta assinatura a mensagem *reqS<sub>k</sub>* é enviada aos coordenadores dos outros sítios.

<sup>10</sup> Na verdade até poderíamos considerar o esquema hierárquico de protocolos como um todo, representando um *Paxos* onde os participantes são trocados na ausência de mensagens dos mesmos em prazos devidos.

Na sequência, o sítio líder deve propor um número de sequência para a requisição  $req_k$ , este número determina a ordem de execução da requisição na ME. O envio do número de sequência da requisição é enviado numa mensagem *propose*. Com a recepção pelos sítios do *propose* envolve o envio, por todos os sítios, uma mensagem de *accept<sub>k</sub>* confirmando a aceitação da proposta de ordem do líder para a requisição  $req_k$ .

No passo final, depois da execução de  $req_k$  na ME, o sítio que recebeu originalmente a requisição assina a uma mensagem de *reply* com os resultados e repassa a mesma para todas as suas réplicas na configuração de  $f + 1$  réplicas, que enviam esta mensagem ao cliente. Basta que uma só destas mensagens chegue ao cliente para que a instância de protocolo se encerre.

Na sequência descrevemos as partes em detalhes (na forma de pequenos algoritmos) deste algoritmo hierárquico.

### 5.2.2 Interações do Cliente com o Serviço Replicado Hierárquico

O esquema hierárquico inicia quando um cliente  $C_i$ , desejando enviar uma requisição de serviço (pedido da execução da operação *op*), usa as informações de um serviço replicado que adquiriu de um repositório de nomes. Estas informações são na verdade endereços *IP* e portas das réplicas coordenadoras dos sítios do sistema. Como dito na seção 5.1.3, as informações de endereço das coordenadoras de sítios do sistema são registradas no repositório considerado por um dado sítio  $S_h$ . Este registro corresponde à visão que  $S_h$  possui sobre réplicas coordenadoras no sistema. Este conhecimento de  $S_h$  sobre as coordenadoras de cada sítio é representado nos nossos algoritmos como o *array coord<sub>h</sub>*. Cada sítio registra seu *coord<sub>h</sub>* (o seu conhecimento do sistema) em um repositório disponível em seu domínio de nomes.

As informações contidas em um *array coord<sub>h</sub>* pode apresentar inconsistências, devido às dimensões do sistema e as facilidades de reconfiguração que apresenta os sítios do sistema (trocas de réplicas/*VMs* coordenadoras como mecanismo de tratamento a detecções de intrusões). A melhora da qualidade destas informações depende de atualizações periódicas destas informações nos repositórios usados a partir dos sítios do sistema.

O cliente  $C_i$  querendo enviar a requisição  $req_k$  deve fazer uso das informações de uma lista de coordenadores (*coord<sub>h</sub>*) disponível em um repositório de nomes que tenha acesso. Destes endereços escolhe uma

réplica  $R_{rj}$  como receptora de sua requisição<sup>11</sup>. Esta escolha de réplica pode levar em consideração critérios de proximidade, uma vez que, endereços  $IP$  estão disponíveis em  $coord_h$ . Outro aspecto a ser considerado é que o cliente não sabe a que sítio está enviando a sua requisição (se é o sítio líder ou não) e muito menos se a réplica receptora é uma coordenadora de seu sítio. As reconfigurações dinâmicas (trocas de coordenadores e de sítios líderes) que podem ocorrer no sistema contribuem para este fato. Mas, mesmo assim, a requisição  $req_k$  é encaminhada para a réplica  $R_{rj}$  escolhida.

Esta escolha é representada no algoritmo 1 pelo uso do índice  $j$  (linha 3) que é usado para percorrer o array  $coord_h$ . O envio da requisição é ativado na linha 4 do mesmo algoritmo, no momento em que o código de aplicação ativa um *RequestforOperation* (), passando a operação  $op$  que deseja executar na replicação Máquina de Estado do serviço alvo.

**Algoritmo 1.** Interações do Cliente  $C_i$  com o Serviço Replicado

**Init**

- |  |  |
|--|--|
| <ol style="list-style-type: none"> <li>1. <math>n_c \leftarrow 0</math>;</li> <li>2. <math>prazoClient \leftarrow initialValue</math>;</li> <li>3. <math>j \leftarrow 1</math>;</li> <li>4. <b>upon</b> <i>RequestforOperation</i>(<math>op</math>):</li> <li>5. <math>n_c = n_c + 1</math>;</li> <li>6. <math>id_m \leftarrow \langle id_{ci}, n_c \rangle</math>;</li> <li>7. <math>Sign_{ci} \leftarrow Signature \langle REQUEST, id_m, op \rangle, D_{ci}</math>;</li> <li>8. <math>req_k \leftarrow \langle REQUEST, id_m, op, Sign_{ci}, Cert_{ci} \rangle</math>;</li> <li>9. <b>send</b> <math>req_k</math> <b>to</b> <math>R_{rj} : id_{rj} = coord_h[j]</math>;</li> <li>10. <math>\delta_{req\_nc.id} \leftarrow id_m</math>;</li> <li>11. <math>\delta_{req\_nc.time} \leftarrow timer()</math>;</li> </ol> | <ol style="list-style-type: none"> <li>12. <b>upon receive</b> (<math>reply_k</math>) <b>from</b> <math>R_{rj}</math> <b>do</b></li> <li>13. <b>if</b> <math>verifySign(reply_k, Sign_{ASr}, reply_k, Cert_{ASr})</math> <b>then</b></li> <li>14.     <b>if</b> <math>\delta_{req\_nc.id} = reply_k.id_m</math> <b>then</b></li> <li>15.         <math>delivery(reply_k, resp)</math>;</li> <li>16.         <math>\delta_{req\_nc.id} \leftarrow null</math>;</li> <li>17.         <math>\delta_{req\_nc.time} \leftarrow \perp</math>;</li> <li>18.     <b>end if</b></li> <li>19. <b>end if</b></li> <li>20. <b>upon</b> (<math>timer() - \delta_{req\_nc.time} \geq prazoClient</math>) <b>at</b> <math>C_i</math> <b>do</b></li> <li>21.     <math>j \leftarrow j + 1</math>;</li> <li>22.     <b>send</b> <math>req_k</math> <b>to</b> <math>R_{rj} : id_{rj} = coord_h[j]</math>;</li> <li>23.     <math>prazoClient \leftarrow Estimate(prazoClient)</math>;</li> <li>24.     <math>\delta_{req\_nc.time} \leftarrow timer()</math>;</li> </ol> |
|--|--|

A requisição, antes de ser encaminhada, tem o seu número de sequência ( $n_c$ ) e o identificador da requisição determinados (linhas 5-6). O identificador do cliente ( $id_{ci}$ ) e o número de sequência ( $n_c$ ) quando concatenados formam o identificador da requisição do cliente ( $id_m$ , linha 6). Na linha 7, o tipo (*REQUEST*) e o  $id_m$  da requisição, juntos com a operação desejada ( $op$ ) são assinados com a chave privada do cliente ( $D_{ci}$ ). A requisição  $req_k$ , na sua forma final é montada, ao serem agregados a assinatura ( $Sign_{ci}$ ) e o certificado do cliente ( $Cert_{ci}$ ), na linha 8. A mensagem com a requisição é enviada à réplica  $R_{rj}$  que o cliente supõe como sendo coordenadora de um sítio  $S_j$  (linha 9). O cliente ativa a

<sup>11</sup> No nosso modelo, as réplicas através de suas *VMs* são os únicos componentes visíveis através da rede.

estrutura  $\delta_{req\_nc}$  usada no controle de *timeout* ao associar a mesma à requisição  $req_k$  (linha 10) e registrando o tempo de envio da mensagem em campo apropriado desta estrutura (linha 11).

As duas *threads* subsequentes do algoritmo representam a recepção de resposta que veio do serviço replicado (linhas 12 – 19) e o tratamento do não atendimento do prazo estipulado para o *timeout* *prazoClient* (linhas 20 – 24).

No caso da recepção, a assinatura da resposta ( $reply_k$ ) é verificada na linha 13, usando o certificado do sítio  $S_j$ <sup>12</sup>. A partir da identificação da mensagem (linha 14), os resultados da execução no servidor replicado da operação *op* são entregues para a aplicação cliente (linha 15). Depois, a estrutura de *timeout* ( $\delta_{req\_nc}$ ) é dissociada de  $req_k$  e seu registro de tempo é desativado (linhas 16-17). Como são enviadas respostas de várias réplicas do mesmo sítio de serviço ( $f + 1$   $reply_k$ 's) para o cliente, a primeira que chega é processada e a dissociação da estrutura de *timeout* faz com que as restantes que cheguem depois sejam descartadas.

O tratamento do prazo ultrapassado (*prazoClient*) nas linhas 20-24 simplesmente envolve o reenvio da requisição não respondida. Mas, desta vez, para outra réplica contida em  $coord_h$  de outro sítio. O incremento de  $j$  sinaliza o envio para outra réplica do serviço alvo (linha 21). A estratégia que assumimos neste trabalho é de que a cada exceção o cliente envia para outra réplica contida em  $coord_h$ . O cliente fica percorrendo o *array* citado até que a resposta correspondente à requisição  $req_k$  seja recebida pelo mesmo. Outras estratégias poderiam ser assumidas. Por exemplo, enviar para todas réplicas em  $coord_h$  simultaneamente, mas esta estratégia poderia representar em custo excessivo ao cliente dependendo das dimensões do sistema.

Nas linhas 23 e 24, uma nova estimativa do prazo a ser usado é definido com a função *Estimate()* que aumenta o prazo anterior com um acréscimo  $\Delta$  unidades de tempo no valor deste intervalo de tempo (linha 23). A estrutura é reativada na linha 24 com um novo valor correspondendo ao novo tempo de (re)envio da mensagem de  $req_k$ .

O envio de assinaturas nas mensagens trocadas, com os respectivos certificados dos assinantes, evita ataques conhecidos como de *Sybil* [Newsome, et al., 2004] e *man-in-the-middle* [Murthy e Manoj, 2004] no esquema apresentado.

---

<sup>12</sup> O certificado (chave pública) usado é do *Agreement Service (AS<sub>j</sub>)* do sítio  $S_j$ , que é quem assina a mensagem resposta ( $reply_k$ ) enviada ao cliente  $C_i$ .

### 5.2.3 Protocolo de Admissão de Requisições de Clientes

Uma mensagem de requisição de cliente ( $req_k$ ) é admitida no sistema para execução quando um dos *Agreement Services* do sistema (elementos confiáveis) assina esta requisição. O algoritmo 2 representa estas trocas para admissão de uma requisição de cliente. Uma réplica  $R_{rj}$  do sítio  $S_j$  recebe esta mensagem e repassa a mesma para o *Agreement Service*  $AS_j$  do sítio, através de operação de memória compartilhada (linha 2).

Algoritmo 2. Protocolo de admissão de requisição de cliente ( $req_k$ ) no sistema

<pre> {Replica R<sub>rj</sub>} 1. upon receive (req<sub>k</sub>) at R<sub>rj</sub> do 2.   shared_mem.msgBuffer<sub>rj</sub>.write(req<sub>k</sub>); 3. upon msgSignal at R<sub>rj</sub>: coord<sub>rj</sub>[j]= id<sub>rj</sub> do 4.   msg ← shared_mem.protBuffer.read(); 5.   if msg.type = REQS then 6.     reqS<sub>rj</sub> ← msg; 7.     reqS<sub>rj</sub>.id ← id<sub>rj</sub>; 8.     send reqS<sub>rj</sub> to all S<sub>i</sub>; 9.   end if 10. upon msgSignal at R<sub>rj</sub> do 11.   msg ← shared_mem.protBuffer.read(); 12.   if msg.type = REPLY then 13.     reply<sub>rj</sub> ← msg; 14.     reply<sub>rj</sub>.id ← id<sub>rj</sub>; 15.     send reply<sub>rj</sub> to C<sub>i</sub>; 16.   end if </pre>	<pre> {Agreement Service AS<sub>j</sub>} 17. upon msgSignal at AS<sub>j</sub> do 18.   msg ← shared_mem.msgBuffer<sub>rj</sub>.read(); 19.   if msg.type = REQUEST then 20.     if verifySign (msg.Sign, msg.Cert) then 21.       reply<sub>k</sub> ← replyCache.read(msg.id<sub>m</sub>); 22.       if reply<sub>k</sub> ≠ null then 23.         shared_mem.protBuffer.write (reply<sub>k</sub>); 24.       else 25.         if reqSCache(msg.id<sub>m</sub>) = null then 26.           reqCache<sub>j</sub> ← reqCache<sub>j</sub> ∪ {req<sub>k</sub>}; 27.           Sign<sub>AS<sub>j</sub></sub> ← Signature (&lt;msg.id<sub>m</sub>, req<sub>k</sub>&gt;, D<sub>AS<sub>j</sub></sub>); 28.           reqS<sub>k</sub> ← &lt;REQS, null, msg.id<sub>m</sub>, req<sub>k</sub>, Sign<sub>AS<sub>j</sub></sub>, Cert<sub>AS<sub>j</sub></sub>&gt;; 29.           shared_mem.protBuffer.write (reqS<sub>k</sub>); 30.           reqSCache<sub>j</sub> ← reqSCache<sub>j</sub> ∪ {msg}; 31.           δ<sub>prot<sub>j</sub></sub>.id ← id<sub>m</sub>; 32.           δ<sub>prot<sub>j</sub></sub>.time ← timer(); 33.           int ← 1; 34.         end if 35.       end if 36.     end if 37.   end if </pre>
--	--

Quando o *Agreement Service* ( $AS_j$ ) recebe uma sinalização de presença de mensagem em *buffer* compartilhado de imediato a leitura da mesma é executada (linhas 17-18, algoritmo 2). Nas linhas 19-20, é verificado se a mensagem é do tipo *REQUEST* e se a assinatura do cliente da mesma é válida. Na sequência, outro teste inspeciona se a operação transportada nesta mensagem já foi executada e, portanto, se existe um *reply* correspondente em *cache* (linhas 21-22 do algoritmo 2). Se existir o *reply<sub>k</sub>* com os resultados da operação, então o *reply<sub>k</sub>* é encaminhado para as réplicas do sítio ( $R_{rj}$ ) via memória compartilhada (*shared\_mem.protBuffer*) para que estas enviem então a resposta ao cliente correspondente (linha 23).

Se não estiver presente em *cache* o *reply<sub>k</sub>*, um novo teste é realizado (linha 25). O teste verifica se esta requisição já foi admitida no sistema. Se a mesma não foi ainda admitida, a requisição  $req_k$  recebida é

então armazenada em *cache* (*reqCache<sub>j</sub>*) e, na sequência, é assinada pelo *AS<sub>j</sub>* usando sua chave privada *D<sub>AS<sub>j</sub></sub>* (linhas 26-27). Uma mensagem de requisição assinada (*reqS<sub>k</sub>*) é montada com o campo do identificador do futuro emissor ainda não preenchido (*null*). São passadas também outras informações como o identificador da requisição (*id<sub>m</sub>*), a requisição do cliente (*req<sub>k</sub>*), a assinatura obtida (*Sign<sub>AS<sub>j</sub></sub>*) e o certificado do *AS<sub>j</sub>* (*Cert<sub>AS<sub>j</sub></sub>*, linha 28). Esta mensagem *reqS<sub>k</sub>* é encaminhada via memória compartilhada (*shared\_mem.protBuffer*) à réplica coordenadora do sítio *S<sub>j</sub>* (linha 29). A *reqS<sub>k</sub>* (requisição do cliente assinada) é ainda salva também em *cache* (*reqSCache<sub>j</sub>*) na linha 30.

Por fim, o *AS<sub>j</sub>* inicia estrutura de *timeout* ( $\delta_{prot\_1}$ ) com o identificador da requisição (*id<sub>m</sub>*) e o início do prazo (*prazoCoord*) para receber resposta do sítio líder (linhas 31-32). Um contador de trocas de visão (variável *int*) foi criado em nossa proposta para indicar o número de trocas de visão local no sítio *S<sub>j</sub>* durante uma instância de protocolo. Ou seja, o valor de *int* corresponde ao número de trocas de coordenador durante a execução do protocolo de acordo (*Paxos*) que define a ordem de execução de uma requisição *req<sub>k</sub>* de um cliente. A cada início de instância de protocolo o valor de *int* é inicializado (linha 33). Este contador será descrito em detalhes na seção 5.4.

Ainda no algoritmo 2, o coordenador *R<sub>rj</sub>* recebe uma sinalização do *AS<sub>j</sub>* de mensagem em memória compartilhada (linha 3). Nas linhas 4 e 5, *R<sub>rj</sub>* recebe a mensagem e verifica se a mesma é do tipo *REQS*. Assim sendo, prepara a mensagem *reqS<sub>rj</sub>* adicionando o seu identificador (*id<sub>rj</sub>*) como emissor e envia a mesma para os outros sítios do sistema (linhas 6-8).

A requisição assinada *reqS<sub>k</sub>* quando recebida é a prova de sua admissão para execução no sistema. Os coordenadores de outros sítios ao receberem esta mensagem que terão a requisição do cliente endossada (assinada) por um componente confiável (*AS<sub>j</sub>*) do sistema.

Nas linhas 10-16 do algoritmo 2, é mostrado o envio de uma mensagem *reply<sub>k</sub>* pelas réplicas do sítio *S<sub>j</sub>*. Este procedimento é o mesmo também apresentado no algoritmo 6. Uma réplica *R<sub>rj</sub>* é sinalizada na linha 10 pela presença de mensagem em memória compartilhada. Nas linhas 11-12, ocorrem a leitura da mensagem e a identificação da mesma como sendo do tipo *REPLY*. Neste caso, a réplica inclui sua identificação como emissora e envia a mensagem final *reply<sub>rj</sub>* ao cliente que tinha enviado a requisição *req<sub>k</sub>* correspondente (linhas 13- 15 do algoritmo 2). Todas as réplicas *R<sub>rj</sub>* de *S<sub>j</sub>* devem enviar copias do *reply<sub>k</sub>*. Neste sentido, como trabalhamos com o limite de *f* réplicas maliciosas, qualquer comportamento malicioso de réplica será sempre mascarado no cliente.

### 5.2.4 Processamento de Requisições Admitidas

O algoritmo 3 contém os passos que envolvem a definição de um valor de número na ordem global da execução das requisições de clientes pelas réplicas dos diversos sítios que formam o esquema hierárquico proposto. A mensagem de requisição do cliente já assinada pelo *Agreement Service* do sítio visível pelo cliente (ou seja,  $reqS_{rj}$ ) é recebida pelos coordenadores  $R_{rj}$ 's e repassada para os seus respectivos  $AS_j$ 's nas linhas 1 e 2 do algoritmo 3.

**Algoritmo 3.** Processamento de mensagem requisição assinada ( $reqS_k$ ) no sistema.

<pre> {Replica R<sub>rj</sub>} 1. upon receive (reqS<sub>rj</sub>) at R<sub>rj</sub>: coord<sub>rj</sub>[j]=id<sub>rj</sub> do 2.  shared_mem.msgBuffer<sub>rj</sub>.write (reqS<sub>k</sub>); 3.  upon msgSignal at R<sub>rj</sub>: coord<sub>rj</sub>[l]=id<sub>rj</sub> do 4.    msg ← shared_mem.protBuffer.read(); 5.    if msg.type = PROPOSE then 6.      propose<sub>rl</sub> ← msg; 7.      propose<sub>rl</sub>.id ← id<sub>rl</sub>; 8.      send propose<sub>rl</sub> to all S<sub>i</sub>; 9.    end if </pre>	<pre> {Agreement Service AS<sub>j</sub>} 19. upon msgSignal at AS<sub>j</sub> do 20.  msg ← shared_mem.msgBuffer<sub>rj</sub>.read (); 21.  if msg.type = REQS then 22.    req<sub>k</sub> ← msg.req; 23.    reply<sub>k</sub> ← replyCache<sub>rj</sub>.read(req<sub>k</sub>.id<sub>m</sub>); 24.    if (reqS<sub>k</sub> ∈ reqSCache<sub>rj</sub> ∧ reply<sub>k</sub> = null) then 25.      if verifySign (msg.Sign, msg.Cert) ∧ verifySign (req<sub>k</sub>.Sign, req<sub>k</sub>.Cert) then 26.        reqSCache<sub>rj</sub> ← reqSCache<sub>rj</sub> ∪ {msg}; 27.        δ<sub>prot<sub>rj</sub></sub>.id ← id<sub>m</sub>; 28.        δ<sub>prot<sub>rj</sub></sub>.time ← timer(); 29.        int ← 1; 30.        if AS<sub>j</sub> = AS<sub>l</sub> then 31.          gOrder<sub>l</sub> ← gOrder<sub>l</sub> + 1; 32.          orderedReq<sub>k</sub> ← &lt; req<sub>k</sub>, gOrder<sub>l</sub> &gt;; 33.          orderList<sub>l</sub> ← orderList<sub>l</sub> ∪ { orderedReq<sub>k</sub> }; 34.          Sign<sub>AS<sub>j</sub></sub> ← Signature(&lt; PROPOSE, reqS<sub>k</sub>.id<sub>m</sub>, gOrder<sub>l</sub> &gt;, D<sub>AS<sub>j</sub></sub>); 35.          propose<sub>k</sub> ← &lt; PROPOSE, null, reqS<sub>k</sub>.id<sub>m</sub>, gOrder<sub>l</sub>, Sign<sub>AS<sub>j</sub></sub>, Cert<sub>AS<sub>j</sub></sub> &gt;; 36.          shared_mem.protBuffer.write (propose<sub>k</sub>); 37.        end if 38.      end if 39.    end if 40.  end if </pre>
---	---

Neste caso, o *Agreement Service* de um sítio  $S_j$  recebe um sinal de que a mensagem está na memória compartilhada, lê e verifica como sendo do tipo *REQS* (linhas 19-21). Nesta situação, o  $AS_j$  recupera a requisição do cliente ( $req_k$ ) na mensagem *REQS* e busca em cache o *reply* correspondente (linha 22-23). Na sequência, nas linhas 24 e 25, é testado se já houve a recepção (se recebeu  $reqS_k$ ) e a execução da requisição (se  $reply_k \neq null$ ) e ainda, se as assinaturas na mensagem são válidas. A verificação de assinaturas envolve a verificação das assinaturas de  $reqS_k$  (requisição do cliente assinada pelo *Agreement Service* que a colocou no sistema) e de  $req_k$  (assinatura do cliente da requisição). Se a mensagem é correta, não foi recebida e também não foi executada, a mesma ( $reqS_k$ ) é colocada em cache (linha 26). Com isto, a identificação da mensagem e o



tempo de chegada são registradas em uma estrutura ( $\delta_{prot\_1}$ ) do  $AS_j$  (linhas 27-28).

Na linha 29, é também iniciado o contador de trocas de visão local (variável *int*, detalhes na seção 5.4) nos sítios do sistema receptores de  $reqS_k$ . Ou seja, todos os sítios vão ter no controle da variável *int* a indicação de quantas trocas de visão local realizaram na execução da instância  $id_m$  do protocolo de acordo.

Se o sítio  $S_j$  é o líder do nível global ( $AS_j = AS_l$ ), o mesmo deve preparar então uma mensagem de *PROPOSE*, com a sugestão de um número de ordem global ( $gOrder_l$ ) para a requisição  $req_k$  (linhas entre 30 e 36 do algoritmo 3). A ordem atribuída mais informações sobre a requisição do cliente são assinadas pelo  $AS_l$  do sítio líder. Esta mensagem ( $propose_k$ ) é encaminhada pelo  $AS_l$  via memória para a sua réplica  $R_{rl}$  (no sítio líder  $S_l$ ). A mesma ordem definida para  $req_k$  é também incluída em uma lista apropriada ( $orderList_l$ ) em  $AS_l$ . A réplica coordenadora  $R_{rl}$  recebe na memória a mensagem do tipo *PROPOSE*, acrescenta seu identificador como emissor desta mensagem e envia a mesma para todos os outros sítios do sistema (linhas 3-9), finalizando o algoritmo 3.

### 5.2.5 Processamento de Mensagens *PROPOSE*

O algoritmo 4 mostra a preparação das mensagens de *ACCEPT* que definem a aceitação da requisição e a sua ordem no fluxo de execuções dos serviços replicados dos sítios. Uma réplica coordenadora  $R_{rl}$  recebe a mensagem  $propose_{rl}$  que vem do sítio líder  $S_l$  e repassa ao seu *Agreement Service*  $AS_j$  via memória compartilhada (linhas 1 e 2, do algoritmo 4). Entre as linhas 10 e 13, o  $AS_j$  recebe a mensagem, verifica o tipo (*PROPOSE*) e a correção da mensagem (a assinatura é validada). Na sequência, verifica nas linhas entre 14 e 16, se a requisição ( $reqS_k$ ) foi recebida anteriormente e se a mesma não foi executada ( $reply_k = null$ ). Sendo conhecida e ainda não executada, o  $AS_j$  testa também se possui o número global de ordem ( $gOrder_k$ , linhas 17-18) definido pelo sítio líder para a requisição do cliente ( $req_k$ ). Se não tiver coloca em caches a mensagem  $propose_{rl}$  e o número de ordem  $gOrder_k$  atribuído à requisição citada (linhas 19-20).

Entre as linhas 21 e 23, o  $AS_j$  procura a estrutura de controle de prazo ( $\delta_{prot\_1}$ ) para desativar o mecanismo de *timeout* relacionado com a espera da mensagem de *PROPOSE*. Esta estrutura é o mecanismo usado para detectar coordenadores maliciosos (ou sítios líderes maliciosos em nível do *Paxos*) na primeira parte do protocolo. Quando ocorrem estas detecções, mudanças de visões local ou global podem ser provocadas

(mudanças de réplicas coordenadoras em nível local e troca de sítio líder em nível global).

Algoritmo 4. Processamento das mensagens *PROPOSE*

```

{Replica Rj}
1. upon receive (proposei) at Rj : coordj[j]= idj do
2.   shared_mem.msgBufferj.write (proposei);

3. upon msgSignal at Rj : coordj[j]= idj do
4.   msg ← shared_mem.protBuffer.read ();
5.   if msg.type = ACCEPT then
6.     acceptj ← msg;
7.     acceptj.id ← idj ;
8.     send acceptj to all Si;
9.   end if

{Agreement Service ASj}
10. upon msgSignal at ASj do
11.   msg ← shared_mem.msgBufferj.read ();
12.   if (msg.type = PROPOSE) ∧ (msg ∉ proposeCache) then
13.     if verifySign (msg, Sign, msg, Cert) then
14.       reqSk ← reqSCachej.read (msg.idm);
15.       replyk ← replyCachej.read (msg.idm);
16.       if (reqSk ≠ null ∧ replyk = null) then
17.         gOrderk ← orderListj.getOrder (msg.idm);
18.         if (gOrderk = null) then
19.           orderListj ← orderListj ∪ { < reqSk.req, msg.gOrder > };
20.           proposeCachej ← proposeCachej ∪ {msg};
21.           if msg.idm = δprot_1.id then
22.             δprot_1.time ← ⊥;
23.           end if
24.         end if
25.         SignASj ← Signature (<ACCEPT, msg.idm, msg.gOrder>, DASj);
26.         acceptk ← <ACCEPT, null, msg.idm, msg.gOrder, SignASj, CertASj>;
27.         shared_mem.protBuffer.write(acceptk);
28.         δprot_2.id ← idm;
29.         δprot_2.time ← timer();
30.         δprot_2.sync ← 1;
31.       end if
32.     end if
33.   end if

```

Depois, entre as linhas 25 e 26, é montada e assinada uma mensagem de *ACCEPT* que define a concordância do Sítio  $S_j$  sobre a proposição de ordem do líder. A mensagem  $accept_k$  apresenta campos como o *id* da requisição do cliente ( $id_m$ ), a ordem atribuída a esta requisição ( $gOrder_k$ ) e a assinatura e o certificado do  $AS_j$ . A mensagem  $accept_k$  é então passada a réplica coordenadora de  $S_j$  através de memória compartilhada (linha 27).

Uma segunda estrutura de *timeout* ( $\delta_{prot\_2}$ ) é definida para a segunda parte do protocolo de acordo, delimitando desta forma a recepção das mensagens de *accept* trocadas entre os sítios do sistema. A identificação da instância de protocolo em execução (ou seja, o  $id_m$  da requisição do cliente) (linhas 28) e o início do prazo (*prazoAccept*) para receber as mensagens de *accept* (linhas 29) são registrados nesta estrutura  $\delta_{prot\_2}$ . Na linha 30, um campo *sync* é usado para diferenciar no tratamento das temporizações definidas para esta segunda fase. No caso de temporização associada com a recepção de mensagens *accept<sub>k</sub>'s*, este campo *sync* tem armazenado o valor “1”. As discussões sobre as diferentes temporizações nesta segunda parte do algoritmo são apresentadas na seção 5.4 deste texto.

A coordenadora  $R_{rj}$  recebe a mensagem  $accept_k$  enviada pelo  $AS_j$  de seu sítio (linhas 3-5), inclui o seu identificador ( $id_{rj}$ ) na mensagem se definindo como emissor da mesma (linhas 6 e 7) e envia esta mensagem para os coordenadores de todos os sítios do sistema (linha 8 do algoritmo 4).

### 5.2.6 Processamento de Mensagens *ACCEPT*

O algoritmo 5 dá sequência à execução do esquema proposto, envolvendo a aceitação da ordem global da requisição proposta ( $req_k$ ) e a preparação da sua execução pelas réplicas da máquina de estado do sítio considerado. Neste sentido, quando a réplica coordenadora  $R_{rj}$  recebe uma mensagem de  $accept_{ri}$  de um sítio  $S_i$  repassa a mesma para o seu *Agreement Service* (linhas 1-2).

Ao receber um sinal sobre a sua memória compartilhada,  $AS_j$  lê a mensagem, verifica o seu tipo e também se não é mensagem antiga, ou seja, se a mesma não se encontra em nenhuma das *caches* de mensagens  $accept_k$  (linhas 24-26). Na sequência, a assinatura é verificada (linha 27). Sendo nova, do tipo *ACCEPT* e com assinatura validada, a mensagem é então armazenada em *cache* apropriada (linha 28).

Uma vez armazenada a mensagem, é verificado se o total de mensagens  $accept_k$  recebidas dos diferentes sítios atingiu o limite  $g + 1$  (linha 29). Depois, nas linhas 30 e 31, com o uso do identificador da requisição do cliente ( $id_m$ ), é verificado se a mensagem de admissão da requisição ( $reqS_k$ ) já foi recebida por  $S_j$ .

Possuindo a admissão no sistema da requisição (possuindo  $reqS_k$ ) e sendo o número de mensagens  $accept_k$ 's válidas e relacionadas com a instância de acordo de identificador  $id_m$  maior ou igual ao limite  $g + 1$ , o consenso sobre a ordem global proposta para  $req_k$  pelo sítio líder está garantido. Ou seja, o número de sítios aceitando a ordem proposta para  $req_k$  é suficiente ( $|acceptCache_j(msg.id_m)| \geq g+1$ ) para garantir a evolução do esquema segundo o algoritmo *Paxos*. Com isto, a requisição  $req_k$  e sua correspondente ordem global  $gOrder_k$  são recuperadas da lista  $orderList_j$  (linhas 32-33) e colocadas como um par ordenado  $\langle req_k, gOrder_k \rangle$  no *buffer reqReadyCache\_j* (linha 34). Deste *buffer*, as requisições são retiradas segundo a ordem global ( $gOrder$ ) e enviadas às réplicas de  $S_j$  para serem executadas.

Algoritmo 5. Processamento de mensagens *ACCEPT* no Sítio  $S_j$ 

```

{Replica  $R_{ij}$ }
1. upon receive (accept $_i$ ) at  $R_{ij}$ : coord $_{ij}[j]=id_j$  to
2. shared_mem.msgBuffer $_{ij}$ .write (accept $_i$ );

3. upon dataSignal at  $R_{ij}$  do
4. data $_i \leftarrow$  shared_mem.buffer_toExecute.read(nextExecution $_i$ );
5. req $_k \leftarrow$  data $_i$ .req;
6. resp  $\leftarrow$  appThread.execute (req $_k$ .op);
7. response $_j \leftarrow$  <RESPONSE,  $R_{ij}$ .id, req $_k$ .id $_m$ , resp>;
8. nextExecution $_j \leftarrow$  nextExecution $_j + 1$ ;
9. if data $_i$ .gOrder = checkpointNumber $_{ij}$  then
10. checkpoint (checkpointNumber $_{ij}$ );
11. checkpointNumber $_{ij} \leftarrow$  checkpointNumber $_{ij} +$  nextPoint;
12. end if
13. shared_mem.appBuffer $_{ij}$ .write(response $_j$ );

14. upon msgSignal at  $R_{ij}$  do
15. msg  $\leftarrow$  shared_mem.protBuffer.read ();
16. if msg.type = NEW_LOCAL_VIEW then
17. coord $_j[j] \leftarrow$  msg.coord;
18. if id $_j =$  coord $_j[j]$  then
19. newLocalView $_j \leftarrow$  msg;
20. newLocalView $_j$ .id  $\leftarrow$  id $_j$ ;
21. send newLocalView $_j$  to all  $S_i$ ;
22. end if
23. end if

{Agreement Service  $AS_j$ }
24. upon msgSignal at  $AS_j$  do
25. msg  $\leftarrow$  shared_mem.msgBuffer $_j$ .read ();
26. if (msg.type = ACCEPT)  $\wedge$  (msg  $\notin$  acceptCache $_j$  []) then
27. if verifySign (msg.Sign, msg.Cert) then
28. acceptCache $_j$  (msg.id $_m$ )  $\leftarrow$  acceptCache $_j$ (msg.id $_m$ )  $\cup$  {msg};
29. if |acceptCache $_j$ (msg.id $_m$ )|  $\geq$  g + 1 then
30. reqS $_k \leftarrow$  reqSCache $_j$ .read (msg.id $_m$ );
31. if (reqS $_k \neq$  null) then
32. req $_k \leftarrow$  orderList $_j$ .getReq (msg.id $_m$ );
33. gOrder $_k \leftarrow$  orderList $_j$ .getOrder(msg.id $_m$ );
34. reqReadyCache $_j \leftarrow$  reqReadyCache $_j \cup$  (<req $_k$ , gOrder $_k$ >);
35. if accept $_i$ .id $_m = \delta_{prot\_2}$ .id then
36.  $\delta_{prot\_2}$ .temp  $\leftarrow$  1;
37.
38. end if
39. req $_k \leftarrow$  reqReadyCache $_j$ .getReq(nextExecution $_j$ );
40. if req $_k \neq$  null then
41. shared_mem.buffer_toExecute.write(<req $_k$ , nextExecution $_j$ >);
42.  $\delta_{msg}$ .temp  $\leftarrow$  timer ();
43.  $\delta_{msg}$ .id  $\leftarrow$  accept $_i$ .id $_m$ ;
44. end if
45. else
46. coord $_j[j] \leftarrow$  new_coord (list_replicas $_j$ , int);
47.  $L_j \leftarrow L_j + 1$ ;
48. sync  $\leftarrow$  1;
49. Sign $_{AS_j} \leftarrow$  Signature (<NEW_LOCAL_VIEW,
50. accept $_i$ .id $_m$ , coord $_j[j]$ , sync,  $L_j$ , G>,  $D_{AS_j}$ );
51. newLocalView $_k \leftarrow$  <NEW_LOCAL_VIEW,
52. null, accept $_i$ .id $_m$ , coord $_j[j]$ , sync,  $L_j$ , G, Sign $_{AS_j}$ , Cert $_{AS_j}$ >;
53. shared_mem.protBuffer.write(newLocalView $_k$ );
54.  $\delta_{prot\_j}$ .id  $\leftarrow$  accept $_i$ .id $_m$ ;
55.  $\delta_{prot\_j}$ .time  $\leftarrow$  timer ();
56. end if
57. end if
58. end if
59. end if

```

Nas linhas 35-36, é desarmado o controle de *timeout* ( $\delta_{prot\_2}$ ) correspondente à segunda fase do *Paxos*. Este controle de prazo (*prazoAccept*) define o tempo limite para a chegada das mensagens *accept $_k$*  correspondentes à instância de protocolo de identificador *id $_m$* , dentro do limite necessário.

Na linha 38, é iniciada a busca da requisição que tenha ordem correspondente à próxima execução da replicação do sítio  $S_j$ . O ponteiro *nextExecution $_j$* , em nível do *Agreement Service*, indica o número de ordem global da próxima requisição a ser executada em  $S_j$ . Depois da busca em *reqReadyCache $_j$* , um teste é executado (linha 39) para verificar se a requisição procurada (com este número de ordem) já se encontra neste *buffer*. Se a requisição *req $_k$*  com número de ordem global *nextExecution $_j$*  está presente no *buffer* indicado, a mesma é passada via memória compartilhada (*shared\_mem.buffer\_toExecute.write*) para ser executada nas réplicas do sítio  $S_j$  (linha 40). Devido ao não sincronismo

das réplicas de um sítio, as mensagens acessadas por réplicas na memória *shared\_mem.buffer\_toExecute* continuam com cópias nesta memória. Mensagens só são removidas deste *buffer* quando *checkpoints* são executados (veja na seção 5.3).

Em seguida, neste algoritmo, é preparada uma estrutura de *timeout* ( $\tilde{\delta}_{int}$  que chamamos de estrutura de *timeout* interno), cuja finalidade é estabelecer um limite para a execução da requisição  $req_k$  e detectar possíveis comportamentos maliciosos de réplicas no sítio  $S_j$  (linhas 41 e 42 do algoritmo 5). O limite  $timeout_{int}$  leva em consideração a aplicação em execução nas réplicas da máquina de estado do sítio  $S_j$ <sup>13</sup>.

A condição *else* (linha 44) correspondente ao teste da linha 31 que verifica se  $S_j$  possui a mensagem de admissão da instância de protocolo  $id_m$ . Neste caso, como pelo menos  $g + 1$  sítios acordaram sobre a ordenação desta requisição  $req_k$  e  $S_j$  não recebeu a admissão da mesma, o  $AS_j$  suspeita de sua coordenadora local ( $R_{rj}$ ) e prepara uma troca de visão local com a escolha de uma nova coordenadora substituindo a réplica sob suspeita. A troca de visão local é feita com a função *new\_coord* (*list\_replicas<sub>j</sub>*, *int*) na linha 45.

Com o novo coordenador determinado, é necessário incrementar o número de visão local ( $L_j$ ) (linhas 46). Este número de visão local  $L_j$  identifica as visões de um sítio  $S_j$  e também evolui como o contador *int* durante trocas de visão local. Mas diferente deste último,  $L_j$  evolui entre diferentes instâncias do protocolo, sendo sempre crescente.

Uma vez que houve troca de coordenador em  $S_j$ , é necessário enviar informações sobre este novo coordenador para os outros sítios do sistema. Diante disto, uma mensagem assinada de *newLocalView<sub>j</sub>* é montada nas linhas entre 48 e 49 para a propagação da informação de novo coordenador (*coord<sub>j</sub>[j]*) de  $S_j$  no sistema. Esta mensagem leva uma indicação de que haverá envios de mensagens de certificados referentes a esta troca de visão local para a atualização das réplicas de  $S_j$  (linha 47) (*sync = 1*). A mensagem *newLocalView<sub>j</sub>* é repassada para a nova coordenadora  $R_{rj}$  de  $S_j$  para envio aos outros sítios do sistema (linha 50).

---

<sup>13</sup> Considerarmos o *tempMaxExec* como uma estimativa do tempo máximo de execução de uma operação de aplicação nas réplicas e ainda, *tempCheckpointing* como uma estimativa de tempo máximo para a execução da função *checkpoint()*. Neste caso, o chamado *timeout* interno que é o prazo para executar uma requisição da aplicação dever ter valor tal que  $prazoInterno \geq tempMaxExec + tempCheckpointing$ . A função *checkpoint()* será introduzida na sequência.

O fato de  $S_j$  não ter recebido a mensagem  $reqS_k$  e de não ter registro da mensagem de  $propose_k$  correspondente pode também ter sido por congestionamento da rede. Nas linhas entre 51 e 52, é então instanciada uma estrutura  $timeout_{\delta_{prot\_1}}$  que deve controlar o recebimento destas mensagens relacionadas com a primeira fase do protocolo de acordo.

Dando sequência na execução do algoritmo, as réplicas  $R_{rj}$  do sítio  $S_j$  são sinalizadas no algoritmo 5 para receberem em seus respectivos *buffers* de execução a mensagem  $data_k$  enviada pelo  $AS_j$  (linha 3 e 4). A leitura de mensagens nestes *buffers* é feita com ponteiro de ordem global  $nextExecution_{rj}$  (as mensagens não são retiradas destes *buffers* mas copiadas). A requisição  $req_k$  é recuperada da mensagem recebida ( $data_k$ ) (linha 5) e executada em *thread* de aplicação na Réplica  $R_{rj}$ . A execução da  $req_k$  pela réplica gera os resultados que são mantidos em *resp* (linha 6). Estes resultados, na linha 7, são colocados na estrutura  $response_{rj}$ , com os respectivos identificadores da réplica executante ( $R_{rj}.id$ ) e da requisição de cliente executada ( $req_k.id_m$ ). Na sequência, na linha 8, é incrementado o ponteiro  $nextExecution_{rj}$  que controla as execuções da réplica  $R_{rj}$  e segue a ordem global das requisições de clientes.

Se a ordem global da requisição executada corresponde ao próximo *checkpoint*, então o mesmo é executado entre as linhas 9 e 12 do algoritmo 5. Os aspectos que envolvem os *checkpoints* da aplicação sendo executada pelas réplicas, serão retomados na seção seguinte. Na sequência, a mensagem  $response_{rj}$  é então enviada ao *Agreement Service* do sítio  $S_j$  na linha 13 do algoritmo 5.

O procedimento de envio da mensagem  $newLocalView_{rj}$  está entre as linhas 14 e 23 e é o mesmo mostrado no algoritmo 8. Na linha 14, a réplica  $R_{rj}$  recebe a sinalização de mensagem. Lê e verifica o tipo como sendo  $NEW\_LOCAL\_VIEW$  nas linhas 15 e 16. Na sequência, registra o novo coordenador em  $coord_{rj}[j]$  (linha 17). Se  $R_{rj}$  for o novo coordenador, então como tal deve preparar a mensagem  $newLocalView_{rj}$ , colocando-se como emissor desta mensagem (linhas 18-20). Na linha 21, envia  $newLocalView_{rj}$  para todos os sítios  $S_i$  que formam o nível global (na verdade envia para os coordenadores destes sítios). Os detalhes sobre troca de visões estão na seção 5.4.

### 5.2.7 Processamento de Mensagens **RESPONSE**

No algoritmo 6, o  $AS_j$  recebe as mensagens  $response_k$  das réplicas do sítio  $S_j$  e faz a comparação dos resultados de processamento das mesmas (compara os campos  $response_k.resp$  incluídos nas mesmas). Como consequência desta comparação, o  $AS_j$  deve preparar uma mensagem  $reply_k$  que comportará o resultado majoritário como resposta

do processamento da requisição  $req_k$ . Cada réplica  $R_{rj}$  da Máquina de Estados de  $S_j$  comunica, através de mensagens  $response_k$ , seus resultados referentes ao processamento de  $req_k$  ao  $AS_j$  via memória compartilhada (linhas 18-19 do algoritmo 6).

**Algoritmo 6.** Processamento de mensagens  $response_k$  e envio do  $reply$  correspondente ao Cliente

```

{Replica Rrj}
1. upon msgSignal at Rrj do
2.   msg ← shared_mem.protBuffer.read();
3.   if msg.type = REPLY then
4.     replyrj ← msg;
5.     replyrj.id ← idrj;
6.     send replyrj to Ci;
7.   end if
8.   upon msgSignal at Rrj do
9.     msg ← shared_mem.protBuffer.read();
10.    if msg.type = NEW_LOCAL_VIEW then
11.      coordrj[j] ← msg.coord;
12.      if idrj = coordrj[j] then
13.        newLocalViewrj ← msg;
14.        newLocalViewrj.id ← idrj;
15.        send newLocalViewrj to all Si;
16.      end if
17.    end if

{Agreement Service ASj}
18. upon appSignal at ASj do
19.   responsek ← shared_mem.appBufferrj.read();
20.   respListj(responsek.idm) ← respListj(responsek.idm) ∪ { responsek };
21.   if (respListj(responsek.idm) ≥ f + 1) ∨ (timer() - δint.time ≥ timeoutint) then
22.     if match(respListj(responsek.idm), responsek.resp) = f + 1 then
23.       SignASj ← Signature(<REPLY, responsek.idm, responsek.resp>, DASj);
24.       reply ← <REPLY, null, responsek.idm, responsek.resp, SignASj, CertASj>;
25.       replyCachej ← replyCachej.write(reply);
26.       if reqk ∈ reqCachej then
27.         shared_mem.protBuffer.write(reply);
28.       end if
29.       δint.time ← ⊥;
30.       nextExecutionrj ← nextExecutionrj + 1;
31.       if restore = true then
32.         list_replicasj ← restoreVMs (f + 1, idm);
33.         if coordrj[j] ∉ list_replicasj then
34.           coordrj[j] ← new_coord (list_replicasj, int);
35.           Lj ← Lj + 1;
36.           sync ← null;
37.           SignASj ← Signature (<NEW_LOCAL_VIEW,
38.                                responsek.idm, coordrj[j], sync, Lj, G>, DASj);
39.           newLocalViewk ← <NEW_LOCAL_VIEW, null,
40.                                responsek.idm, coordrj[j], sync, Lj, G, SignASj, CertASj>;
41.           shared_mem.protBuffer.write(newLocalViewk);
42.           δpraz_2.id ← responsek.idm;
43.           δpraz_2.time ← timer();
44.           δpraz_2.sync ← null;
45.         end if
46.         restore ← false;
47.       end if
48.     else
49.       if restore = false then
50.         list_replicasj ← initiatedVMs (f, responsek.idm);
51.         restore ← true;
52.         δint.time ← timer();
53.       end if
54.     end if

```

As mensagens  $response_k$  ligadas à requisição de identificação  $id_m$  são armazenadas na lista correspondente ( $respList_j(response_k.id_m)$ , linha 20). Quando esta lista possui mensagens  $response_k$  em número maior ou igual a  $f + 1$  (recebeu respostas de no mínimo  $f + 1$  réplicas) ou quando o prazo  $timeout_{int}$  ( $timeout$  interno) tenha sido alcançado, os resultados dos

processamentos contidos nestas mensagens  $response_k$ 's ( $response_k.resp$ ) devem passar então por um processo de comparação (linhas 21 e 22).

A diversidade de projeto que usamos basicamente envolve só o uso de diferentes sistemas operacionais. A adoção de somente diferentes sistemas operacionais nas diversas réplicas de um sítio não enfraquece a nossa capacidade de tolerância a intrusões. A nossa crença para tal afirmação está no fato de que os ataques basicamente são realizados via rede passando necessariamente por funcionalidades de sistema operacional. Se assumíssemos uma diversidade mais abrangente envolvendo, por exemplo, os níveis de linguagens e de algoritmos da aplicação, as dificuldades seriam maiores e os resultados não melhorariam significativamente. Por exemplo, a comparação das  $response_k$ 's não poderia ser feita com um votador simples (função  $match()$ , na linha 22). Seria necessário o uso de uma função que tratasse com as diferenças de implementação das réplicas de  $S_j$ . Teríamos então de usar um *adjucator* [Pullum, 1993] para esta comparação.

A condição que assumimos de diversidade só de sistemas operacionais no nosso sistema, permite então o uso da função  $match()$  que indica a maior frequência de repetição de um valor de uma variável em um dado conjunto. Na linha 22, a condição de teste estabelecida é de que, quando aplicada a função  $match()$  no conjunto  $respList_j(response_k.id_m)$ , seja conseguido  $f+1$  valores iguais de  $response_k.resp$ . Ou seja, que  $f+1$   $response_k$  de réplicas do sítio tenham resultados iguais do mesmo processamento nas diferentes réplicas de  $S_j$ . Esta condição uma vez verificada, nas linhas 23 e 24, provoca a montagem e assinatura da mensagem de  $reply_k$  pelo  $AS_j$ . Uma cópia desta mensagem é colocada em *cache* na linha 25 (memorização usada para suprir as respostas a reenvios subsequentes de  $req_k$  pelo cliente). Se  $S_j$  foi quem recebeu a requisição original do cliente ( $req_k$ ), antes da mesma ser admitida no sistema, então  $S_j$  deve ser o responsável pelo envio do  $reply_k$  através de suas réplicas ao cliente. O  $AS_j$  determina se o sítio  $S_j$  é quem deve responder ao cliente pelo teste da linha 26 (ou seja, verifica se possui a mensagem original do cliente em  $reqCache_j$ ), como consequência passa para suas réplicas  $R_{rj}$ 's via memória o  $reply_k$  (linha 27). O *timeout* interno é desarmado na linha 29. Por sua vez, o ponteiro  $nextExecution_j$  que indica a próxima requisição a ser executada no esquema (seguindo a ordem global) é incrementado na linha 30.

O esquema proposto inicia com uma configuração de  $f+1$  réplicas que, durante o processamento de uma requisição, devido à ocorrência de até  $f$  réplicas maliciosas, pode ter sua configuração alterada, com acréscimo de  $f$  novas réplicas, para outra de  $2f+1$  réplicas, de modo a



garantir o resultado correto da máquina de estados (ME). Se ocorrer em réplicas maliciosas durante o processamento de uma requisição, esta mudança de configuração é sempre concretizada. Mas, no final do acordo sobre a requisição  $req_k$  e do processamento da mesma, a configuração da replicação ME deve retornar ao número de  $f+1$  réplicas, eliminando às  $f$  possíveis maliciosas. Ou seja, no final do processamento temos um tratamento das réplicas maliciosas, recuperando a configuração inicial. Qualquer requisição deve começar com uma configuração de  $f+1$  réplicas. As demais  $f$  réplicas só são adicionadas se houver qualquer desacordo de resposta entre as  $f+1$  réplicas iniciais.

Nos nossos algoritmos, é definida a variável *restore* que é usada para sinalizar as passagens de uma configuração máxima ( $2f+1$  réplicas) para uma mínima ( $f+1$  réplicas). Na linha 31, um teste sobre esta variável verifica se a replicação ME está ou não com configuração máxima. Como o processamento replicado terminou com o acordo sobre o resultado que representa a ME do sítio  $S_j$ , é necessário verificar se este acordo foi conseguido com configuração máxima. Se for este o caso, então antes de passar a próxima execução de requisição, a ME deve retornar a configuração mínima, ficando pronta para iniciar o acordo e o processamento de uma nova requisição.

Nas linhas entre 32 e 45, é explicitada esta restauração de configuração de  $f+1$  réplicas. A função *restoreVMs()* que é usada na restauração, retorna a nova lista de réplicas (*list\_replicas<sub>j</sub>*) de  $S_j$  com configuração mínima (linha 32). A lista *list\_replicas<sub>j</sub>* mantém os *id*'s e as portas e endereços das réplicas da configuração. Na seção 5.5, a função *restoreVMs()* é descrita em seus detalhes.

Esta restauração de configuração mínima pode significar na troca de visão local (mudança de réplica coordenadora em  $S_j$ ). O teste da linha 33 verifica se a réplica coordenadora (*coord<sub>j</sub>[j]*) pertence à nova lista de réplicas de  $S_j$  (*list\_replicas<sub>j</sub>*). Se não pertencer, a função *new\_coord(list\_replicas<sub>j</sub>, int)* é ativada para uma troca de visão local na linha 34 do algoritmo 6. Diante disto, ocorre um incremento no número de visão local ( $L_j$ ) na linha 35 e uma mensagem assinada *newLocalView<sub>k</sub>* é montada nas linhas entre 37 e 38 para a propagação de informações do novo coordenador (*coord<sub>j</sub>[j]*) de  $S_j$  no sistema. Esta mensagem leva uma indicação (*sync = null*) de que não haverá envios de mensagens de certificados referentes a esta troca de visão local para a atualização das réplicas de  $S_j$  (linha 36). Isto é explicado pelo fato que as réplicas  $R_{rj}$  concluíram a instância de protocolo de identificador  $id_m$  (veja detalhes sobre troca de visões nas seções 5.4). Na linha 39, a mensagem *newLocalView<sub>k</sub>* é repassada pelo  $AS_j$  a coordenadora de  $S_j$ .

A mensagem  $newLocalView_{r_j}$  é enviada pela réplica coordenadora  $R_{r_j}$  a seus pares usando o procedimento descrito entre as linhas 8 e 16 deste algoritmo. As ações indicadas são as mesmas descritas no algoritmo anterior (linhas 14 e 23 do algoritmo 5) e são repetidas também no algoritmo 8.

Nas linhas entre 40 e 42, é instanciada uma estrutura  $timeout_{\delta_{prot\_2}}$  que deve controlar o recebimento de no mínimo  $g + 1$  mensagens  $acknewLocal_k$  referentes à mensagem  $newLocalView_{r_j}$ . O registro da variável  $sync$  nesta estrutura de  $timeout$  indica que não haverá troca de certificados e sim simples reconhecimentos ( $\delta_{prot\_2}.sync = null$ , na linha 42). Os reconhecimentos  $acknewLocal_k$  (que têm os seus processamentos descritos na seção 5.4.2.1) são enviados pelos sítios do sistema assegurando que quando atingido o limite mínimo necessário, a nova coordenadora de  $S_j$  pode se considerar devidamente conectada no sistema para participar de futuros acordos da ME. Na linha 44, a variável  $restore$  volta a indicar uma configuração mínima ( $restore \leftarrow false$ ).

Se na linha 21 o prazo  $timeout_{int}$  for atingido sem que se tenha  $f + 1$  réplicas enviando suas  $response_k$ 's ou se na linha 22 não houver um acordo de  $f + 1$  réplicas em suas respostas, o algoritmo é retomado nas linhas 46 e 52 do algoritmo 6. De início, na linha 47, é verificado se a ME já não está executando uma configuração máxima. Então, se não for este o caso, para que se consiga o acordo sobre os resultados das mensagens  $response_k$ , é necessária a efetivação da passagem de uma configuração mínima (configuração sem o acordo) para a configuração máxima de  $2f + 1$  réplicas. Esta passagem é concretizada com a função  $initiatedVMs()$  que retorna também uma nova lista  $list\_replicas_j$  indicando as  $2f + 1$  réplicas de  $S_j$  em configuração máxima (linha 48). A indicação de configuração máxima ( $restore = true$ ) é feita na linha 49 e o  $timeout$  interno é novamente armado na linha 50. Estas ações ativam  $f$  novas réplicas no sítio  $S_j$ , tentando chegar ao acordo necessário de  $f + 1$  réplicas corretas para a evolução do sistema.

Na linha 1 do algoritmo 6, a réplica  $R_{r_j}$  do sítio  $S_j$  é sinalizada pela presença de mensagem de  $reply_k$  na memória compartilhada com o  $AS_j$ . Esta réplica faz então a leitura da mensagem inclui sua identificação como emissora e envia a mensagem final  $reply_{r_j}$  ao cliente que originou a requisição  $req_k$  (linhas 3-6 do algoritmo 6). Todas as réplicas de  $S_j$  devem enviar cópias do  $reply_k$ . Neste sentido, como trabalhamos com o limite  $f$  para o número de réplicas maliciosas, qualquer comportamento malicioso de réplicas de  $S_j$  será sempre mascarado no cliente.

### 5.3 MECANISMOS DE CHECKPOINT

O esquema proposto possui mecanismos definidos para atualização e recuperação de réplicas de cada sítio. Cada réplica controla o momento em que deve salvar seu estado em memória estável (memória não volátil e que garanta a conclusão das escritas). Cada réplica calcula um resumo do seu estado salvo usando uma função criptográfica de *hash*. Estes *hashes* calculados são passados ao *Agreement Service* do sítio para que o mesmo possa determinar por comparação os estados salvos que estão corretos. O *AS* registra em memória estável o valor de resumo como correto o que se mostra com frequência maior ou igual a  $f + 1$  nas comparações. Quando for solicitado um destes estados para a iniciação de uma *VM* de réplica, o estado recebido poderá ser testado tomando como base o resumo (*hash* de estado) armazenado como correto pelo *AS*.

Os mecanismos de *checkpointing* do esquema proposto são apresentados no algoritmo 7. Parte do algoritmo 5 que envolve a chamada de *checkpoint()* é repetida ainda aqui nesta seção. Esta chamada é feita depois da execução da operação da requisição  $req_k$  do cliente (linhas 1-8 do algoritmo 7). Na linha 4, a ordem global da requisição executada ( $data_k.gOrder$ ) é comparada com o número do próximo *checkpoint* ( $checkpointNumber_{r_j}$ ). Se coincidirem os dois números, a função *checkpoint()* é chamada para salvar o estado da réplica do modelo Máquina de Estado do sítio  $S_j$  (linha 5) e, na sequência, o ponteiro do próximo *checkpoint* ( $checkpointNumber_{r_j}$ ) é atualizado com o valor do período definido entre *checkpoints* ( $nextPoint$ ) na linha 6 do algoritmo 7. O período  $nextPoint$  pode ser definido, por exemplo, para salvamentos de estado entre cada 100 execuções de operações requisitadas por cliente.

A função *checkpoint()* que possui como parâmetro de entrada  $checkpointNumber_{r_j}$  está representada entre as linhas 9 e 19 do algoritmo 7. Esta função executada pelas réplicas  $R_{r_j}$  começa, na linha 10, com a réplica chamadora sendo “parada” na sua execução (isto é, a aplicação é parada antes de iniciar nova execução de requisição). Em seguida, o estado da aplicação, os ponteiros de controle (como o ponteiro  $nextExecution_{r_j}$ ) e *buffers* da réplica são armazenados nas linhas 11 e 12, através de *upcalls* ( $applicationState.get(R_{r_j})$  e  $controlExec.get(R_{r_j})$ ). Na linha 13, é então montado o estado da réplica ( $replicaState_{r_j}$ ) envolvendo o estado da aplicação ( $appState_{r_j}$ ), os ponteiros e *buffers* da réplica ( $replicaControl_{r_j}$ ) junto com o identificador da réplica ( $id_{r_j}$ ) e o número do *checkpoint* ( $checkpointNumber_{r_j}$ ). Este estado de réplica é armazenado na sequência no *buffer*  $checkpoint_{r_j}$  que é específico para a réplica  $R_{r_j}$ , localizado em memória estável (linha 14).

Algoritmo 7. Checkpointing em Réplica  $R_j$ 

```

{Replica  $R_j$ }
1.  $req_i \leftarrow data_i.req$ ;
2.  $resp \leftarrow appThread.execute(req_i.op)$ ;
3.  $responses \leftarrow \langle RESPONSE, R_i.id, req_i.id_m, resp \rangle$ ;
4. if  $data_i.gOrder = checkpointNumber_j$  then
5.    $checkpoint(checkpointNumber_j)$ ;
6.    $checkpointNumber_j \leftarrow checkpointNumber_j + nextPoint$ ;
7. end if
8.  $shared\_mem.appBuffer_j.write(responses)$ ;
9. function  $checkpoint(checkpointNumber_j)$ 
10.  $appThread_j.stop()$ ;
11.  $appState_j \leftarrow applicationState.get(R_j)$ ;
12.  $replicaControl_j \leftarrow controlExec.get(R_j)$ ;
13.  $replicaState_j \leftarrow \langle checkpointNumber_j, id_j, replicaControl_j, appState_j \rangle$ ;
14.  $stableMemory\_Checkpoint_j.write(replicaState_j)$ ;
15.  $hash_j.resume \leftarrow calculateHash(replicaState_j)$ ;
16.  $hash_j.number \leftarrow checkpointNumber_j$ ;
17.  $hash_j.nextCheckpoint \leftarrow checkpointNumber_j + nextPoint$ ;
18.  $shared\_mem.appBuffer_j.write(hash_j)$ ;
19.  $appThread_j.resume()$ ;
20. end function

{Agreement Service  $AS_j$ }
21. upon  $appSignal$  at  $AS$  do
22.    $hash_k \leftarrow shared\_mem.appBuffer_j.read()$ ;
23.    $hashList(hash_k.number) \leftarrow hashList(hash_k.number) \cup \{hash_k\}$ ;
24.   if  $(|hashList(hash_k.number)| \geq f+1) \vee (timer() - \delta_{ms,time} \geq timeout_m)$  then
25.     if  $match(hashList(hash_k.number), hash_k) = f+1$  then
26.        $stableMemory\_Hash.write(\langle hash_k.number, hash_k \rangle)$ ;
27.        $updateLists(hash_k.number, nextPoint)$ ;
28.     else if  $restore = false$  then
29.        $list\_replicas \leftarrow initiatedVMs(f, response_k.id_m)$ ;
30.        $restore \leftarrow true$ ;
31.        $\delta_{ms,time} \leftarrow timer()$ ;
32.     end if
33.   end if
34.   function  $updateLists(checkpointNumber_j, nextPoint)$ 
35.      $previousNumber \leftarrow hash_k.number - nextPoint$ ;
36.      $stableMemory\_Hash.remove(previousNumber)$ ;
37.      $stableMemory\_Checkpoint_j.remove(previousNumber)$ ;
38.      $hashList(previousNumber).delete()$ ;
39.     while  $\exists (msg) \in proposeCache_j, msg.gOrder \leq previousNumber$  do
40.        $proposeCache_j.remove(msg.id_m)$ ;
41.        $acceptCache_j.remove(msg.id_m)$ ;
42.        $reqCache_j.remove(msg.id_m)$ ;
43.        $reqSCache_j.remove(msg.id_m)$ ;
44.        $replyCache_j.remove(msg.id_m)$ ;
45.        $orderList_j.removeOrder(msg.id_m)$ ;
46.        $newLocalViewCache.remove(msg.id_m)$ ;
47.        $shared\_mem.buffer\_toExecute.remove(msg.id_m)$ ;
48.        $respList(id_m).delete()$ ;
49.        $ListCerts(id_m, *, *) .delete()$ ;
50.        $ListGC_j(id_m, *) .delete()$ ;
51.        $ListACKs(id_m, *, *) .delete()$ ;
52.     end while
53.   end function

```

O esquema proposto de *checkpoint* funciona com resumos de estado, para tanto na linha 15, é usada uma função criptográfica de *hash* para o cálculo deste resumo do registro de estado da réplica ( $calculateHash(replicaState_j)$ ). Os valores do resumo e o número do *checkpoint* são armazenados então na estrutura  $hash_{rj}$  (linhas 15 e 16). Na linha 17 é calculado o próximo ponto de *checkpoint* que também é registrado na estrutura de  $hash_{rj}$  ( $hash_{rj}.nextCheckpoint$ ). Por fim, esta estrutura  $hash_{rj}$  é repassada para o  $AS_j$  via memória compartilhada (linha 18) e a réplica  $R_{rj}$  é retomada em seu processamento normal (linha 19) e a função retorna na linha 6 do algoritmo 7 (ou linha 10 do algoritmo 5). Na linha 21, o *Agreement Service* do sítio ( $AS_j$ ) é sinalizado pela presença de  $hash_k$  em memória compartilhada. Nas linhas 22-23, lê  $hash_k$  e armazena o mesmo em lista relacionada ao seu número de *checkpoint*. O teste da linha 24 verifica se o número de  $hash_k$ 's recebidos na lista correspondente alcançou o limite de  $f+1$ . Alcançado este número, os resumos de estado das réplicas são comparados para verificar se os

mesmos coincidem (ou seja, se seus estados são iguais) na linha 25. Se  $f + 1$  resumos coincidem, é formado então um par ordenado com o número do *checkpoint* (*checkpointNumber<sub>j</sub>*) e a estrutura de *hash* correspondente (*hash<sub>k</sub>*). Este par é escrito em *buffer* apropriado de memória estável (linha 26). Lembramos que o uso da diversidade na nossa proposta se limita ao nível de sistema operacional e que isto não dificulta o uso da função *match()* sobre resumos de estado de réplicas de um sítio.

Na linha 27, é chamada a função *updateLists()*, tendo como parâmetros o número do *checkpoint* (*checkpointNumber*) e o período entre *checkpoints* (*nextPoint*). Esta função remove o estado e o *hash* correspondentes ao *checkpoint* anterior (linhas entre 36 e 37 do algoritmo 7). A função *updateLists()* remove também a lista de *hashes* do *checkpoint* anterior (linha 38). Por fim, nesta função, entre as linhas 39 e 52, as listas e *cashes* são atualizadas ou removidas, retirando todos os registros das mensagens que possuem ordem global (*gOrder*) menor ou igual ao número do prévio *checkpoint*.

Se no teste da linha 24 o *timeout* interno (*timeout<sub>int</sub>*) for atingido sem que se tenha  $f + 1$  réplicas enviando seus *hash<sub>k</sub>*'s ou se na linha 25 não houver uma coincidência de  $f + 1$  réplicas em seus resumos, o algoritmo é retomado nas linhas 28 e 29 do algoritmo 7, com a passagem para a configuração máxima de  $2f + 1$  réplicas (usando a função *initiatedVMs()*). A indicação de configuração máxima (*restore = true*) é feita na linha 30. E a estrutura do prazo *timeout<sub>int</sub>* é novamente armada na linha 31. Estas ações ativam  $f$  novas réplicas no sítio  $S_j$ , tentando chegar ao acordo necessário de  $f + 1$  réplicas corretas sobre seus valores de resumo (iguais) do *checkpoint* de modo a permitir a evolução do sistema.

## 5.4 TROCAS DE VISÃO

Nesta seção, é tratado o que se instituiu como trocas de visão, ou seja, trocas de coordenadores nos sítios (local) e de sítios líderes no *Paxos* (nível global) no nosso esquema. Estas trocas estão sujeitas a detecções de malícia que são concretizadas na nossa proposta em controles de prazos (controles de *timeout*). O uso de assinaturas em todos os níveis reduz a possibilidade de malícia a uma simples “não entrega de mensagens” dentro de prazos específicos.

O protocolo *Paxos* apresenta duas temporizações: uma que define um prazo para a chegada de mensagem *PROPOSE* enviada pelo sítio líder; e a segunda relacionada com as recepções de *ACCEPT*'s emitidas por todos os sítios que participam do *Paxos*.

### 5.4.1 Temporizações na Primeira Parte do Protocolo

Na primeira parte do protocolo em um sítio  $S_j$ , as temporizações estão associadas com a chegada de mensagem de *propose*. O algoritmo 8 mostra as ações executadas no tratamento das exceções levantadas nos sítios quando neste prazo não são verificadas as condições de chegada previstas destas mensagens de *propose*.

O controle da ultrapassagem do intervalo de tempo *prazoCoord* que tinha sido armado (linhas 27 e 28 do algoritmo 3) está explicitado na linha 19 no algoritmo 8. Este intervalo de tempo (*prazoCoord*) corresponde ao prazo estipulado para a chegada de mensagem *propose<sub>k</sub>* que vem do sítio líder ( $S_j$ ) com a proposta de um número global para a requisição *req<sub>k</sub>* de cliente. Não chegando esta mensagem dentro deste prazo, a *thread* entre as linhas 19 e 40 do algoritmo 8 é ativada no  $AS_j$ . As linhas 20-21 servem de um filtro para o caso da chegada desta mensagem simultaneamente com o disparo do *timeout* correspondente, evitando processamentos desnecessários com trocas para recuperar o sítio  $S_j$ . Basicamente nestas linhas é verificado se a mensagem *propose<sub>k</sub>* já se encontra em *cache*.

Não encontrada esta mensagem em *cache*, na sequência (linha 22), a *thread* testa se a variável *int* que conta o número de ultrapassagens das temporizações definidas no nosso esquema, durante a execução de uma instância do protocolo de acordo. A temporização *prazoCoord* é a única que nos permite detectar a malícia no sítio líder ( $S_l$ ) do sistema.

Se este contador (*int*) tiver o seu valor inferior a  $f + 1$ , é assumido que o problema da não chegada da mensagem está centrado na malícia do coordenador local, que é quem representa o sítio  $S_j$  no *Paxos* (protocolo executado em nível global). Se atingirmos  $f + 1$  ou mais ultrapassagens deste prazo *prazoCoord*, então teremos a situação em que pelo menos uma réplica local correta do sítio  $S_j$  assumiu o papel de coordenadora deste sítio no *Paxos*. Com a persistência do problema, então a atenção passa a se concentrar sobre o sítio líder ( $S_l$ ) do sistema. Ou seja, este líder passa a ser suspeito de um comportamento malicioso por não está enviando devidamente mensagens *propose<sub>k</sub>*. Em outras palavras, na falta desta mensagem, em ultrapassagens de *prazoCoord* inferior a  $f + 1$ , estas exceções provocam trocas de visões locais (trocas de coordenadores no sítio considerado). Quando este número de ultrapassagens em um sítio ( $S_j$ ) atinge o valor  $f + 1$ , pelo menos uma réplica correta assumiu o papel de coordenadora em  $S_j$  e, portanto, o tratamento destas exceções devem envolver trocas de visão global (outro sítio assume o papel de líder no *Paxos*).

Do procedimento descrito teremos então que o *timeout* para o sítio líder (*prazoLider*) assume o valor de  $(f + 1) * \text{prazoCoord}$ . A Figura 5.7 ilustra esta relação.

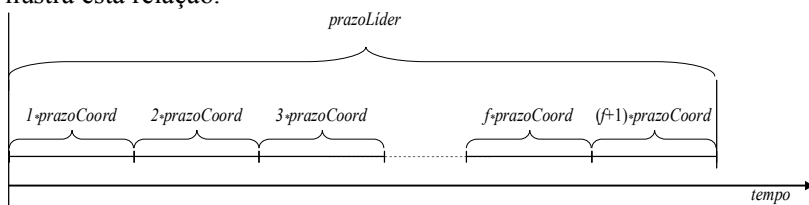


Figura 5.7 – Relação entre *timeouts* no sistema proposto.

Na linha 23 do algoritmo 8, com o contador *int* (contador de trocas visão local) indicando valores inferiores a  $f + 1$ , é iniciado um tratamento de troca de visão local (*local view*). Uma troca de visão local em  $S_j$  implica na mudança de réplica coordenadora deste sítio. Neste caso, a não chegada da mensagem *propose<sub>k</sub>* dentro do *prazoCoord* implica na execução da função *new\_coord(list\_replicas<sub>j</sub>, int)* no sítio  $S_j$ . Os parâmetros *list\_replicas<sub>j</sub>* e *int* são usados na determinação do novo coordenador nesta função que é descrita entre as linhas 41 e 45. Nesta função, o valor de *int* é incrementado e é usado como índice na lista de réplicas (*list\_replicas<sub>j</sub>*) já ordenadas segundo valores de seus identificadores. Ou seja, o próximo coordenador de  $S_j$  será a réplica que ocupa a posição indicada pelo novo valor de *int* ( $int + 1$  da linha 42) na lista *list\_replicas<sub>j</sub>* com os *id*'s destas réplicas já devidamente ordenados. O número da visão local ( $L_j$ ) deve também ser incrementado depois da definição da nova réplica coordenadora (linha 24). As informações da nova réplica coordenadora de  $S_j$  devem então ser propagadas no sistema.

Nas linhas 26 e 27 do algoritmo 8, uma mensagem *newLocalView<sub>j</sub>* é preparada e assinada pelo  $AS_j$ . Esta mensagem comporta informações como o identificador  $id_m$  da requisição de cliente, o novo coordenador de  $S_j$  ( $coord_j[j]$ ), o número da visão local estabelecida ( $L_j$ ) e o número da visão global atual ( $G$ ). É também campo desta mensagem a variável *sync* que indica se serão ou não trocadas mensagens de certificados que provocam a sincronização das réplicas dos sítios do sistema. No caso da linha 25, a variável *sync* tem atribuído o valor "1", indicando que mensagens de certificados serão trocadas no sistema.

A mensagem *newLocalView<sub>j</sub>* é enviada via memória compartilhada para as réplicas  $R_{rj}$  de  $S_j$  (linha 28). Um novo prazo (*prazoCoord*) para a chegada da mensagem *propose<sub>k</sub>* é estimado na linha 29. A estrutura de *timeout*  $\delta_{prot\_1}$  é então reiniciada estabelecendo um novo prazo para recuperar esta mensagem *propose<sub>k</sub>* (linha 30).

Algoritmo 8 – Tratamento de *timeouts* de mensagens PROPOSE

```

{Replica  $R_{rj}$ }                                     {Agreement Service  $AS_j$ }

1.  upon msgSignal at  $R_{rj}$  do                       19. upon (timer() -  $\delta_{prot,j,time}$ )  $\geq$  prazoCoord at  $AS_j$  do
2.  msg  $\leftarrow$  shared_mem.protBuffer.read();      20. propose $_k \leftarrow$  proposeCache.get( $\delta_{prot,j,id_m}$ );
3.  if msg.type = NEW_LOCAL_VIEW then             21. if propose $_k =$  null then
4.  coord $_r[j] \leftarrow$  msg.coord;                22. if int <  $f+1$  then
5.  if id $_r =$  coord $_r[j]$  then                       23. coord $[j] \leftarrow$  new_coord(list_replicas, int);
6.  newLocalView $_{rj} \leftarrow$  msg;                24.  $L_j \leftarrow L_j + 1$ ;
7.  newLocalView $_{rj}.id \leftarrow id_{rj}$ ;          25. sync  $\leftarrow 1$ ;
8.  send newLocalView $_{rj}$  to all  $S_i$ ;              26. Sign $_{AS_j} \leftarrow$  Signature(<NEW_LOCAL_VIEW,
9.  end if                                          $\delta_{prot,j,id_m}, coord[j], sync, L_j, G, >, D_{AS}$ );
10. end if                                         27. newLocalView $\leftarrow$  <NEW_LOCAL_VIEW,
                                                    null,  $\delta_{prot,j,id_m}, coord[j], sync, L_j, G, Sign_{AS_j}, Cert_{AS_j}$ >;
11. upon msgSignal at  $R_{rj}$ : coord $_r[j] = id_{rj}$  do  28. shared_mem.protBuffer.write(newLocalView);
12. msg  $\leftarrow$  shared_mem.protBuffer.read();      29. prazoCoord  $\leftarrow$  Estimate(prazoCoord);
13. if msg.type = GVIEW_CHANGE then              30.  $\delta_{prot,j,time} \leftarrow$  timer();
14. gviewChange $_e \leftarrow$  msg;                  31. else
15. gviewChange $_e.id \leftarrow id_{rj}$ ;            32.  $G \leftarrow G + 1$ ;
16. send gviewChange $_e$  to all  $S_i$ ;                33. sync  $\leftarrow 1$ ;
17. send gviewChange $_e$  to all  $R_{rt}$  at  $S_c$ ;        34. Sign $_{AS_j} \leftarrow$  Signature(<GVIEW_CHANGE, id $_m$ , sync,  $L_j, G, >, D_{AS}$ );
18. end if                                         35. gviewChange $_e \leftarrow$  <GVIEW_CHANGE, null, id $_m$ , sync,  $L_j, G, Sign_{AS_j}, Cert_{AS_j}$ >;
                                                    36. shared_mem.protBuffer.write(gviewChange $_e$ );
                                                    37. prazoCoord  $\leftarrow$  Estimate(prazoCoord);
                                                    38.  $\delta_{prot,j,time} \leftarrow$  timer();
                                                    39. end if
                                                    40. end if
                                                    41. function new_Coord(list_replicas, int)
                                                    42. int  $\leftarrow$  int + 1;
                                                    43. coord  $\leftarrow$  order(list_replicas, int);
                                                    44. return coord;
                                                    45. end function

```

Na linha 1, a réplica  $R_{rj}$  recebe a sinalização de mensagem. Lê e verifica o tipo como sendo *NEW\_LOCAL\_VIEW* nas linhas 2 e 3. Na sequência, registra o novo coordenador em  $coord_{rj}[j]$  (linha 4). Se  $R_{rj}$  for o novo coordenador, então como tal deve preparar a mensagem  $newLocalView_{rj}$ , colocando-se como emissor desta mensagem (linhas 5-7). Na linha 8, envia  $newLocalView_{rj}$  para todos os sítios  $S_i$  que formam o nível global (na verdade envia para os coordenadores destes sítios).

A condição *else* da linha 31 é ativada pela condição  $int \geq f + 1$  (referente ao teste da linha 22 sobre o contador *int* de trocas de visões). O processamento entre estas linhas é provocado porque foi alcançado o prazo do sítio líder ( $(f + 1) * prazoCoord$ ) sem que a mensagem *propose $_k$*  tenha sido recebida pelo sítio  $S_j$ . O problema da não chegada desta mensagem no prazo passa a ser tratado como uma necessidade de troca de *view* global. Como consequência, o número da visão global ( $G$ ) é incrementado na linha 32. O número  $G$  indica a atual visão global do sistema (a evolução deste contador  $G$  é também monotônica e independente da instância de protocolo). Na sequência, é atribuído o valor “1” para a variável *sync* indicando que serão trocadas mensagens de certificados (linha 33).



Nas linhas 34 e 35 do algoritmo 8, é então preparada e assinada uma mensagem  $gviewChange_k$  pelo  $AS_j$ , envolvendo informações como o identificador da requisição de cliente ( $id_m$ ), a variável  $sync$ , o número da visão local atual de  $S_j$  ( $L_j$ ) e o número de visão global com o novo valor estabelecido ( $G$ ). Esta mensagem é enviada via memória compartilhada para a réplica coordenadora que faz a comunicação da mesma para os outros sítios (linha 36). O intervalo de espera ( $prazoCoord$ ) por mensagens  $propose_k$  passa por uma nova estimativa na linha 37. E a estrutura  $\delta_{prot\_1}$  de  $timeout$  é também reiniciada para o novo prazo de recuperação de mensagens  $propose_k$  (linha 38).

Na linha 11, a réplica coordenadora  $R_{rj}$  é sinalizada para ler mensagem em memória compartilhada. O tipo da mensagem é verificado como  $GVIEW\_CHANGE$  na linha 12. Na sequência, a coordenadora prepara a mensagem  $gviewChange_{rj}$ , colocando-se como emissor desta mensagem (linhas 14-15). Na linha 16, envia  $gviewChange_{rj}$  para todos os sítios  $S_i$  que formam o nível global. Também envia  $gviewChange_{rj}$  para todas as réplicas  $R_{ri}$  (do sítio líder) provocando com isto a mudança de coordenadora no sítio líder  $S_i$  (linha 17).

#### 5.4.1.1 Processamento de Mensagem $NEW\_LOCAL\_VIEW$

O algoritmo 9 mostra o processamento de uma mensagem  $newLocalView_{ri}$ . Quando a coordenadora  $R_{rj}$  do sítio  $S_j$  recebe esta mensagem, imediatamente envia a mesma para o seu *Agreement Service* via memória compartilhada (linhas 1 e 2 do algoritmo 9).

O *Agreement Service*  $AS_j$  é sinalizado de mensagem em memória e na sequência lê esta mensagem (linhas 20-21 do algoritmo 9). No teste da linha 22, é verificado se a mensagem é do tipo  $NEW\_LOCAL\_VIEW$  e se não é uma mensagem antiga. Na linha 23, é verificada a assinatura da mensagem e também um filtro para mensagens antigas que ainda não foram recebidas. Este filtro evita processamentos indesejados com mensagens antigas não recebidas anteriormente. Mensagens  $newLocalView_{ri}$  que apresentem  $L_i$  menor que o registro sobre a visão local de  $S_i$  mantida por  $S_j$  ( $(L_j(i) \geq newLocalView_{ri}.L)$ ) são descartadas e o mesmo ocorre para mensagens de troca de visão que apresentem valor de visão global ( $G_i$ ) menor que o mantido por  $S_j$  ( $(G_j(j) > newLocalView_{ri}.G)$ ).

No caso de mensagem nova e válida (enviada pelo sítio  $S_i$ ), o processamento da linha 24 é ativado com o registro feito pelo  $AS_j$  sobre a informação de novo coordenador no sítio  $S_i$  no *array coord<sub>j</sub>* ( $coord_j[i] \leftarrow msg.coord$ ). Na sequência, os números de visão local ( $L_i$ ) e global ( $G_i$ ) do sítio  $S_i$  são também salvos pelo  $AS_j$  (linhas 25 e 26 do algoritmo 9). A

mensagem recebida é então colocada em *cache* (em *newLocalViewCache* na linha 27).

Algoritmo 9. Processamento de mensagem *NEW\_LOCAL\_VIEW*

```

{Replica  $R_{rj}$ }                                     {Agreement Service  $AS_j$ }
1. upon receive (newLocalViewri) at  $R_{rj}$  : coord $r_j$ [ $j$ ] = id $r_j$  to 20. upon msgSignal at  $AS_j$  do
2.   shared_mem.msgBuffer $r_j$ .write (newLocalViewri);
3. upon msgSignal at  $R_{rj}$  : coord $r_j$ [ $j$ ] = id $r_j$  do
4.   msg ← shared_mem.protBuffer.read();
5.   if msg.type = CERTIFICATE_MSG then
6.     coord $r_j$  ← msg.coord;
7.     certMsg $r_j$  ← msg;
8.     certMsg $r_j$ .id ← id $r_j$ ;
9.     send certMsg $r_j$  to all  $S_i$ ;
10.  end if
11. upon msgSignal at  $R_{rj}$  : coord $r_j$ [ $j$ ] = id $r_j$  do
12.   msg ← shared_mem.protBuffer.read();
13.   if msg.type = ACK_NEW_LOCAL_VIEW then
14.     coord $r_j$  ← msg.coord;
15.     acknewLocal $r_j$  ← msg;
16.     acknewLocal $r_j$ .id ← id $r_j$ ;
17.     send acknewLocal $r_j$  to newLocalView $ri$ .id;
18.   end if
19. end if
21. msg ← shared_mem.msgBuffer $r_j$ .read();
22. if msg.type = NEW_LOCAL_VIEW ∧ (msg ≠ newLocalViewCache) then
23.   if verifySign (msg.Sign, msg.Cert) ∧ (L $r_j$ (i) < msg.L) ∧ (G $r_j$ (j) ≤ msg.G) then
24.     coord[i] ← msg.coord;
25.     L $r_j$ (i) ← msg.L;
26.     G $r_j$ (j) ← msg.G;
27.     newLocalViewCache ← newLocalViewCache ∪ {msg};
28.   if (msg.sync ≠ null) then
29.     msgCert.accept ← acceptCache.get (msg.id $m$ );
30.     msgCert.propose ← proposeCache.get (msg.id $m$ );
31.     msgCert.reqS ← reqSCache.get (msg.id $m$ );
32.     msgCert.newLocalView ← newLocalViewCache(msg.id $m$ );
33.     Sign $AS_j$  ← Signature (<CERTIFICATE_MSG, id $m$ ,
34.                           msgCert, list_replicas, coord, sync, L $r_j$ (i), G $r_j$ (j)>, D $AS_j$ );
35.     certMsg $k$  ← <CERTIFICATE_MSG, null, id $m$ ,
36.                           msgCert, list_replicas, coord, sync, L $r_j$ (i), G $r_j$ (j), Sign $AS_j$ , Cert $AS_j$ >;
37.     shared_mem.protBuffer.write(certMsg $k$ );
38.   else
39.     Sign $AS_j$  ← Signature (<ACK_NEW_LOCAL_VIEW,
40.                           msg.id $m$ , list_replicas, msg.coord, msg.sync, msg.L $r_j$ (i), msg.G $r_j$ (j),
41.                           acknewLocal $k$  ← <ACK_NEW_LOCAL_VIEW, null, msg.id $m$ ,
42.                           msg.coord, msg.sync, msg.L $r_j$ (i), msg.G $r_j$ (j), Sign $AS_j$ , Cert $AS_j$ >;
43.     shared_mem.protBuffer.write(acknewLocal $k$ );
44.   end if
45. end if

```

Na linha 28, outro teste é realizado sobre a mensagem. Neste caso, é verificado se haverá ou não troca de certificados entre os sítios do sistema. Na sequência, com a variável *sync* da mensagem indicando a troca de certificados (*newLocalView<sub>ri</sub>*.*sync* = 1), são recuperadas de *cache* mensagens de *accept*, *propose*, *reqS* e de *newLocalView* referentes à requisição de cliente de identificador *id<sub>m</sub>* (linhas entre 29 e 32). Estas informações são alocadas em campos da mensagem *certMsg<sub>k</sub>* que é montada e assinada nas linhas 33 e 34.

São também adicionadas à mensagem *certMsg<sub>k</sub>* as informações das réplicas participando em  $S_j$  (*list\_replicas<sub>j</sub>*), a visão de coordenadoras de  $S_j$  (*array coord<sub>j</sub>*), e os números de visão local ( $L_i$ ) e de visão global ( $G_i$ ) do sítio  $S_i$ . Os valores de  $L_i$  e  $G_i$  servem para ligar os certificados do sítio  $S_j$  à mensagem *newLocalView<sub>ri</sub>* enviada por  $S_i$ . Estes valores servem também para a localização de mensagem *certMsg<sub>k</sub>* em listas apropriadas dos sítios que irão receber esta mensagem. Na linha 35, o  $AS_j$  envia via

memória compartilhada a mensagem  $certMsg_k$  para a réplica coordenadora  $R_{rj}$  de  $S_j$ .

A condição *else* da linha 36 (parte do teste da linha 28) ativa o processamento de mensagens  $newLocalView$  que possuem o campo  $sync$  indicando a não troca de certificados ( $sync = null$ ). Neste sentido, nas linhas 37-38, é montada e assinada uma mensagem de reconhecimento ( $acknewLocal_k$ ) à mensagem  $newLocalView_{r_i}$  enviada anteriormente por  $S_i$ . Esta mensagem  $acknewLocal_k$  é montada com as informações como o  $id_m$  da instância do protocolo, as informações das réplicas de  $S_j$  ( $list\_replicas_j$ ), a visão de coordenadoras do sistema de  $S_j$  ( $array\ coord_j$ ), a variável  $sync$  e os números de visão local ( $L_i$ ) e de visão global ( $G_i$ ) do sítio  $S_i$  (sítio que deve receber este reconhecimento). Esta mensagem  $acknewLocal_k$  é passada para as réplicas  $R_{rj}$  via memória compartilhada na linha 39.

Na linha 3, a réplica  $R_{rj}$  recebe a sinalização de mensagem. A mensagem recebida é verificada como sendo do tipo  $CERTIFICATE\_MSG$  (linhas 4 e 5). Na sequência, registra a lista de coordenadores ( $coord_j$ ) na lista correspondente de  $R_{rj}$  (em  $coord_{rj}$  linha 6). A partir deste momento, a coordenadora de  $S_j$  ( $R_{rj}$ ) fica sabendo da existência de nova coordenadora em  $S_i$ . A mensagem  $certMsg_{rj}$  é então montada com o identificador de  $R_{rj}$  ( $id_{rj}$ ) e enviada para todos os coordenadores de sítios do sistema (linhas 7, 8 e 9).

Em situação oposta, na linha 11, a réplica  $R_{rj}$  recebe também a sinalização de mensagem. Entre as linhas 12 e 13, a mensagem recebida é verificada como sendo do tipo  $ACK\_NEW\_LOCAL\_VIEW$ . Ou seja, neste caso não serão enviados certificados, mas somente um simples reconhecimento. Com isto, a lista de coordenadores mantida por  $S_j$  ( $coord_j$ ) é salva na lista correspondente de  $R_{rj}$  (em  $coord_{rj}$  na linha 14). Neste momento, a réplica coordenadora do sítio  $S_j$  pode enviar este reconhecimento. A mensagem  $acknewLocal_{rj}$  é então montada com o identificador de  $R_{rj}$  ( $id_{rj}$ ). Esta mensagem deve ser enviada somente ao coordenador do sítio  $S_i$  que trocou de visão local (linhas 15, 16 e 17).

#### 5.4.1.2 Processamento de Mensagem $GVIEW\_CHANGE$

O processamento de mensagens  $gviewChange_k$  segue o algoritmo 10 abaixo. Num primeiro instante, um sítio (através de sua coordenadora) recebe uma mensagem de pedido de mudança de visão global de um elemento confiável de outro sítio que já experimentou  $f+1$  mudanças de visão local. Esta mensagem é repassada ao *Agreement Service* do sítio receptor para o devido processamento. O recebimento desta mensagem por réplicas é mostrado no algoritmo 10 em duas partes: primeiro por

réplicas coordenadoras  $R_{rj}$  dos sítios  $S_j$  do sistema (linhas 1-2); e, na segunda situação, por todas as réplicas  $R_{rl}$  do sítio líder  $S_l$  (linhas 3-4) para forçar que a mensagem de troca de visão global chegue ao  $AS_l$ , evitando neste caso a intermediação de uma possível coordenadora maliciosa no sítio líder ( $S_l$ ). Em ambas situações, as réplicas passam via memória a mensagem  $gviewChange_k$  para seus respectivos *Agreement Services*.

O processamento de um  $AS_j$  começa com a leitura e verificação se a mensagem é do tipo  $GVIEW\_CHANGE$  e se já foi recebida anteriormente (linhas 13-15). Na sequência, é verificado se a assinatura da mensagem é válida e é aplicado um filtro ( $G_j(j) \leq msg.G$ ) impedindo o processamento de mensagens  $gviewChange_k$  antigas ainda não recebidas (linha 16). Sendo a mensagem habilitada para o processamento, nas linhas 17 e 18 do algoritmo 10, são armazenados os valores de visão global ( $gviewChange_{ri}.G$ ) e de visão local ( $gviewChange_{ri}.L$ ) do solicitante de troca de visão ( $S_i$ ). Na linha 19, a mensagem recebida é colocada em lista ( $ListGC(gviewChange_{ri}.id_m, G_j(i))$ ) cujo o acesso é determinado pelos valores de  $id_m$  e de  $G_i$  provenientes da mensagem de pedido de troca de visão global.

O teste da linha 20 verifica quando a lista  $ListGC(id_m, G_j(i))$  possui a maioria das mensagens dos  $g + 1$  sítios. Verificada esta situação, o acordo segundo o *Paxos* para a troca de visão global está garantido e, como consequência, o processamento do pedido desta troca de visão global estaria habilitado para prosseguir<sup>14</sup>. Uma vez conseguido este acordo, o sítio  $S_l$  através de seu *Agreement Service* assume o novo valor

---

<sup>14</sup> As nossas premissas de *Agreement Services* confiáveis (e isolados) implicam também a necessidade de um acordo como no *Paxos* original para a troca de visão do sistema (visão global). Embora confiáveis, a não chegada de uma mensagem esperada ( $propose_k$ ) durante  $f + 1$  trocas consecutivas de visões locais pode ter sido provocada por rede congestionada (é bom lembrar que estamos em um sistema que apresenta momentos de assincronismo). E este sítio mandando sua mensagem de troca de visão global, por ser confiável, se fosse processada de imediato provocaria inconsistências no sistema. Por isto o *Paxos* (e a nossa proposta) exige um acordo entre os sítios participantes. Ou seja, para o acordo tem que chegar pelo menos  $g + 1$  mensagens  $gviewChange_k$  dos sítios participantes. Neste caso, teríamos que ter o armazenamento de mensagens e a espera para atingir  $g + 1$  mensagens de mudança de visão global (mensagens com o mesmo valor de  $G$ ). O mesmo ocorre com as mensagens  $certMsg_k$  no algoritmo 10.

de número de visão global ( $G_j(j) \leftarrow gviewChange.G$ ), na linha 21 do algoritmo 10.

**Algoritmo 10.** Processamento de Mensagem *GVIEW\_CHANGE*

```

{Replica Rij}                                     {Agreement Service ASj}
1.  upon receive (gviewChangeg) at Rij : coordij[j] = idij do
2.  shared_mem.msgBufferij.write (gviewChangeg);
3.  upon receive (gviewChangeg) at Rij : idij ∈ list_replicasg
4.  shared_mem.msgBufferij.write (gviewChangeg);
5.  upon msgSignal at Rij : coordij[j] = idij do
6.  msg ← shared_mem.protBuffer.read();
7.  if msg.type = CERTIFICATE_MSG then
8.  coordij ← msg.coord;
9.  certMsgij ← msg;
10. certMsgij.id ← idij;
11. send certMsgij to all Si;
12. end if
13. upon msgSignal at ASj do
14. msg ← shared_mem.msgBufferij.read();
15. if msg.type = GVIEW_CHANGE ∧ (msg ∉ ListGC) then
16. if verifySign (msg.Sign, msg.Cert) ∧ (Gj(j) ≤ msg.G) then
17.   Gj(j) ← msg.G;
18.   Lj(j) ← msg.L;
19.   ListGC (msg.idm, Gj(j)) ← ListGC(msg.idm, Gj(j)) ∪ {msg};
20.   if |ListGC (msg.idm, Gj(j))| ≥ g + 1 then
21.     Gj(j) ← msg.G;
22.     Sl ← new_siteLider(Sites_list, gviewChange.G);
23.     msgCert.accept ← acceptCache.get (msg.idm);
24.     msgCert.propose ← proposeCache.get (msg.idm);
25.     msgCert.reqS ← reqSCache.get (msg.idm);
26.     msgCert.newLocalView ← newLocalViewCache(msg.idm);
27.     if ASj = ASl then
28.       coord[j] ← new_coord (list_replicasg, int);
29.     end if
30.     SignASj ← Signature (<CERTIFICATE_MSG, idm,
31.                          msgCert, list_replicasg, coordij, Lij, G, DASj);
32.     certMsgij ← < CERTIFICATE_MSG,
33.                  null, idm, msgCert, list_replicasg, coordij, Lij, G, SignASj, CertASj
34.                  shared_mem.protBuffer.write(certMsgij);
35.   end if
36. end if
37. end if
38. end if
39. procedure new_siteLider (Sites_list, G)
40.   Sl ← order (Sites_list, G);
41.   return Sl;
42. end proc

```

De posse do novo número de visão global ( $G$ ), o novo sítio líder (novo  $S_l$  do sistema) é determinado pelo  $AS_j$  (linha 22). Na sequência, a definição do novo sítio líder ocorre com o uso da função  $new\_siteLider()$  que tem como parâmetros a lista com os sítios do sistema ( $Sites\_list$ ) e o valor do novo  $G$ . Nesta função que está descrita entre as linhas 36 e 39 do algoritmo 10, o valor do novo número de visão global ( $G$ ) é usado como índice na lista de sítios ( $Sites\_list$ ) ordenada segundo valores dos identificadores de seus sítios no sistema. Ou seja, o próximo líder  $S_l$  será o sítio que ocupa a posição indicada pelo novo  $G$  nesta lista.

Na sequência, são recuperadas de *cache*'s as mensagens de *accept*, *propose*, *reqS* e de *newLocalView* associadas à requisição de cliente com identificador  $id_m$  (linhas entre 23 e 26). Estas informações recuperadas são alocadas em campos da mensagem  $certMsg_k$  que é montada e assinada nas linhas 30 e 31. Antes, se  $S_j$  for o sítio líder ( $S_j$  desempenhando o papel

de  $S_i$ ), a chegada de mensagens de  $gviewChange_k$  no líder questionado deve levar o seu  $AS_i$  a executar a função  $new\_coord(list\_replicas_j, int)$ , provocando a troca do seu coordenador (linhas 27-28). Ou seja, o *Agreement Service* do líder questionado desconfia de seu coordenador (na recepção de um pedido de troca de visão global) e troca o mesmo por outra de suas réplicas.

Na montagem da mensagem de certificados ( $certMsg_k$ ) das linhas 30 e 31, são também adicionadas informações das réplicas participando em  $S_j$  ( $list\_replicas_j$ ), a visão de coordenadoras de  $S_j$  ( $array\ coord_j$ ) e o número da visão global ( $G_j$ ). O valor de  $G_j$  serve para ligar os certificados de  $S_j$  à mensagem  $gviewChange_k$  enviadas pelos outros sítios do sistema. Como número de visão local registrado em campo da mensagem ( $certMsg_k.L$ ) está o valor  $L_M$ . Este valor é usado como um limite máximo para números de visão local. A dificuldade de vincular os diversos certificados trocados com os diferentes números de visão local dos vários emissores de pedidos de troca global (emissores das possíveis  $2g + 1$  mensagens  $gviewChange_k$ ) leva ao uso de  $L_M$  como valor único representando este grupo de emissores<sup>15</sup>. Na linha 32, o  $AS_j$  envia via memória compartilhada a mensagem  $certMsg_k$  para a réplica coordenadora  $R_{rj}$  de  $S_j$ .

Entre as linhas 5 e 12, é descrita a atuação à nível das réplicas coordenadoras dos sítios quando recebem as mensagens do tipo *CERTIFICATE\_MSG* de seus *Agreement Services*. Este procedimento é o mesmo já descrito no algoritmo 9 (seção 5.4.1.1). Também ocorre o registro da lista de coordenadores ( $coord_j$ ) na lista correspondente de  $R_{rj}$  (em  $coord_{rj}$  linha 8 do algoritmo 10). A mensagem  $certMsg_{rj}$  é então montada com o identificador da coordenadora  $R_{rj}$  ( $id_{rj}$ ) e enviada para todos os coordenadores de sítios do sistema (linhas 9, 10 e 11).

As mensagens de certificados montadas pelos  $AS_j$ 's do sistema, como resultado do processamento das mensagens recebidas de  $gviewChange_k$ , são idênticas em formato àquelas geradas a partir de  $newLocalView_k$  no algoritmo 9 (apresentado na seção 5.4.1.1) e são processadas também pelo algoritmo 11 que processa os certificados enviados no sistema (apresentado na seção 5.4.1.3) gerando as

---

<sup>15</sup> As mensagens de certificado não alteram os registros dos sítios receptores. Portanto atribuir este valor máximo ( $L_M$ ) não vai provocar inconsistências no sistema. Ao mesmo tempo, este valor máximo permite mensagens de certificados associadas à troca de visão global passarem por filtros que usam valores normais de números de visão local (valores de  $L$ ).

atualizações correspondentes de listas e variáveis e ainda a sincronização entre os diferentes sítios. Estas mensagens de  $gviewChange_k$  e os certificados correspondentes não desviam significativamente do que é feito no *Paxos*.

#### 5.4.1.3 Processamento de Certificados

O Algoritmo 11 mostra a recepção de mensagens de certificados de uma coordenadora do sítio  $S_j$  (linha 1 do algoritmo citado). Na sequência, a coordenadora  $R_{rj}$  passa a mensagem  $certMsg_k$  para o seu *Agreement Service* ( $AS_j$ ). O  $AS_j$  é ativado com a sinalização de mensagem em memória (linha 26). A mensagem é verificada como do tipo  $CERTIFICATE\_MSG$  e como ainda não recebida (linha 28). A validação de assinatura e a efetivação de um filtro ( $L_j(i) \leq certMsg_k.L \wedge G_j(j) \leq certMsg_k.G$ ) que evita mensagens antigas não recebidas anteriormente, são concretizados através do teste da linha 29.

Depois, uma sequência de operações recupera informações da mensagem de certificados (linhas 30-31). É recuperada a lista de réplicas do sítio emissor da mensagem ( $list\_replicas_k$  na linha 30) e também a lista de coordenadores que este sítio emissor possuía ( $array\ coord_k$ , linha 31). Na linha 32, uma cópia da mensagem  $certMsg_k$  é incluída em lista de certificados apropriada ( $ListCert(certMsg_k.id_m, L_i, G_i)$ ). Os valores de  $L_i$ ,  $G_i$  e o  $id_m$  ( $id$  da última transação de cliente), retirados da mensagem  $certMsg_k$ , ligam esta mensagem de certificados ao sítio ( $S_i$ ) que enviou uma mensagem de troca de visão. Estes valores são também usados como índices para o acesso à lista de certificado correspondente.

A seguir, um teste verifica se já ocorreram  $g + 1$  ou mais mensagens  $certMsg_k$ 's na lista  $ListCert(certMsg_k.id_m, certMsg_k.L, certMsg_k.G)$  (linha 33). Uma vez atingido este limite de mensagens, a mudança de visão proposta e acordada por mensagens de certificados é assumida. Nas linhas entre 34 e 37, são então retiradas as mensagens de certificados da lista apropriada, uma a uma, e usadas na montagem das correspondentes mensagens de recuperação ( $msgRec_k$ ). Cada mensagem de recuperação possui informações de certificados ( $certMsg_k.msgCert$ ), lista de réplicas ( $list\_replicas_k$ ) e coordenador ( $coord_j[k]$ ) de cada sítio emissor dos certificados ( $certMsg_k$ ). As mensagens  $msgRec_k$  são enviadas para as réplicas  $R_{rj}$  do sítio  $S_j$  para que estas atualizem suas informações sobre os sítios do sistema (linha 39).

Também no algoritmo 11, entre as linhas 3 e 5, a réplica  $R_{rj}$  é sinalizada e lê mensagem de memória compartilhada que é, por sua vez, verificada como sendo do tipo  $MSG\_REC$  (mensagem de recuperação enviada pelo  $AS_j$ ). Nesta condição, os certificados contidos em campo

desta mensagem ( $msg.msgCert$ ) são salvos em *certificates*, na linha 6. A mensagem também é usada para atualizar o conhecimento da réplica em relação ao coordenador ( $coord_j [i]$ ), referente ao sítio emissor da mensagem de certificados (no caso  $S_i$ , na linha 7).

Depois, na linha 8, outra condição é verificada: se existe nos certificados recebidos a requisição do cliente, assinada como prova de admissão da mesma ao sistema (*certificates.reqS*), ou seja, se os certificados incluem mensagem do tipo *REQS*. Verificado a existência desta requisição assinada (admitida no sistema) a mesma é recuperada em  $reqS_k$  (linha 9). Em seguida, na linha 10 do algoritmo 11,  $reqS_k$  é enviada ao  $AS_j$  (*Agreement Service* do sítio  $S_j$ ) via memória compartilhada para o processamento adequado (procedimentos do algoritmo 3).

Algoritmo 11. Processamento de Certificados

<pre> {Replica R<sub>rj</sub>} 1. upon receive (certMsg<sub>i</sub>) at R<sub>rj</sub>: coord<sub>rj</sub>[j]=id<sub>rj</sub> to 2.   shared_mem.msgBuffer<sub>rj</sub>.write (certMsg<sub>i</sub>); 3. upon msgSignal at R<sub>rj</sub> do 4.   msg ← shared_mem.protBuffer.read(); 5.   if msg.type = MSG_REC then 6.     certificates ← msg.msgCert; 7.     coord<sub>rj</sub>[i] ← msg.coord; 8.     if certificates.reqS ≠ null then 9.       reqS<sub>k</sub> ← certificates.reqS; 10.      shared_mem.msgBuffer<sub>rj</sub>.write (reqS<sub>k</sub>); 11.    end if 12.    if certificates.propose ≠ null then 13.      propose<sub>k</sub> ← certificates.propose; 14.      shared_mem.msgBuffer<sub>rj</sub>.write (propose<sub>k</sub>); 15.    end if 16.    if certificates.accept ≠ null then 17.      accept<sub>k</sub> ← certificates.accept; 18.      shared_mem.msgBuffer<sub>rj</sub>.write (accept<sub>k</sub>); 19.    end if 20.    if certificates.newLocalView ≠ null then 21.      newLocalView<sub>k</sub> ← certificates.newLocalView; 22.      shared_mem.msgBuffer<sub>rj</sub>.write(newLocalView<sub>k</sub>); 23.    end if 24.  end if </pre>	<pre> {Agreement Service AS<sub>j</sub>} 26. upon msgSignal at AS<sub>j</sub> do 27.   msg ← shared_mem.msgBuffer<sub>rj</sub>.read (); 28.   if msg.type = CERTIFICATE_MSG ∧ msg ∉ ListCert[ ] then 29.     if verifySign (msg.Sign, msg.Cert) ∧ (L(i) ≤ msg.L ∧ G(j) ≤ msg.G) then 30.       list_replicas ← msg.list_replicas; 31.       coord<sub>i</sub> ← msg.coord; 32.       ListCert (msg.id<sub>m</sub>, msg.L, msg.G) ← ListCert (msg.id<sub>m</sub>, msg.L, msg.G) ∪ {msg}; 33.       if  ListCert (msg.id<sub>m</sub>, msg.L, msg.G)  ≥ g+1 then 34.         while ∃(certMsg) ∈ ListCert (msg.id<sub>m</sub>, msg.L, msg.G) do 35.           msgRec<sub>i</sub> ← &lt; MSG_REC, coord<sub>rj</sub>[i], certMsg.id<sub>m</sub>, certMsg.msgCert &gt;; 36.           shared_mem.protBuffer.write (msgRec<sub>i</sub>); 37.         end while 38.       end if 39.     end if 40.   end if </pre>
---	---

Na linha 12, é verificado, nestes mesmos certificados recebidos, se existe mensagem de tipo *PROPOSE* (*certificates.propose*). Existindo a mesma, esta mensagem é recuperada em  $propose_k$  (linha 13). Na sequência, na linha 14 do algoritmo,  $propose_k$  é enviada ao  $AS_j$  para o processamento correspondente (procedimentos do algoritmo 4). Entre as linhas 16 e 19, ocorre procedimento idêntico ao anterior, entretanto desta vez, é para a recuperação de mensagem do tipo *ACCEPT*. O envio é



também para o  $AS_j$  para o processamento da mesma segundo procedimento correspondente (o algoritmo 5).

Entre as linhas 20 e 24, o ocorre o mesmo processo de recuperação, mas desta vez são mensagens  $newLocalView_k$ . Isto é também importante, porque sítios que recebem mensagens de certificados são atualizados mesmo sem ter recebido a mensagem correspondente de troca de visão. O recebimento na recuperação de um sítio de uma mensagem  $newLocalView_k$  não recebida anteriormente pode fazer com que este sítio emita seus certificados e ajude a destravar outros sítios.

Estas trocas generalizadas de certificados e recuperações descritas acima, nesta e nas outras seções (5.4.1, 5.4.1.1, 5.4.1.2), servem para o sítio que perdeu a mensagem  $propose_k$  referente à instância protocolo de identificador  $id_m$ , possa se sincronizar com os demais sítios do sistema. Mas, estas mesmas trocas podem também sincronizar novamente todas as réplicas de todos os sítios do sistema permitindo que estas recuperem também outras mensagens e listas deste protocolo.

## 5.4.2 Temporizações na Segunda Parte do Protocolo

As trocas de mensagens de *accept* no *Paxos* definem uma segunda temporização (um segundo tipo de *timeout*) que atua na segunda parte do protocolo. No nosso esquema, esta segunda temporização também serve para detectar malícia nas trocas em nível de sítios. Uma vez que as mensagens são assinadas no nosso esquema, as únicas ações maliciosas possíveis dos participantes se resumem ao não envio de mensagens que podem ser modeladas a nível global como falhas de *crash* do *Paxos* original. Mas as temporizações desta segunda parte do protocolo não conseguem individualizar ações maliciosas de um sítio líder, uma vez que as trocas de mensagens são generalizadas.

Estas temporizações da segunda parte do nosso protocolo envolvem, basicamente, mensagens de *accept* e de reconhecimento a trocas de visões locais.

### 5.4.2.1 Mensagens ACK\_NEW\_LOCAL\_VIEW

A restauração de configuração mínima descrita no algoritmo 6 (seção 5.2.7) pode levar a situações onde a replica coordenadora é excluída da nova configuração. Estas situações implicam na troca de visão local (escolha de nova coordenadora no sítio considerado) e na geração de mensagens correspondentes de  $newLocalView_k$ . Situações como estas são muito especiais, porque a instância de protocolo  $id_m$  é concluída corretamente (mas em configuração máxima). A troca de certificados subsequente a uma mensagem de troca de visão local nestas condições é

completamente desnecessária, uma vez que, o sítio que emite a mensagem  $newLocalView_k$  possui suas estruturas devidamente atualizadas. Diante disto, só existe a necessidade de propagar o novo coordenador desta nova configuração mínima.

O algoritmo 9 (seção 5.4.1.1) descreve o processamento destas mensagens de troca visão local ( $newLocalView_k$ ) que não envolvem trocas de certificados. Como resultado deste processamento, os outros sítios devem enviar mensagens de reconhecimento ( $acknewLocal_k$ ) para o sítio  $S_j$  que trocou de visão local e não precisa de certificados para se atualizar (linhas 37-40 do algoritmo 9). Estas mensagens  $acknewLocal_k$  quando recebidas no limite mínimo de  $g + 1$  dão a garantia ao sítio  $S_j$  que sua nova coordenadora  $R_{rj}$  é devidamente conhecida no sistema.

O algoritmo 12 apresentado nesta seção trata com o processamento destas mensagens de reconhecimento ( $acknewLocal_k$ ). Nas linhas 1-2, a réplica coordenadora de  $S_j$  recebe uma destas mensagens de reconhecimento e repassa a mesma via memória compartilhada para o seu *Agreement Service*.

**Algoritmo 12.** Processamento de mensagem  $ACK\_NEW\_LOCAL\_VIEW$

<pre> {Replica R<sub>rj</sub>} 1. upon receive (acknewLocal<sub>k</sub>) at R<sub>rj</sub>: coord<sub>rj</sub>[j] = id<sub>rj</sub> to 2.   shared_mem.msgBuffer<sub>rj</sub>.write (newLocalView<sub>k</sub>); </pre>	<pre> {Agreement Service AS<sub>j</sub>} 13. upon msgSignal at AS<sub>j</sub> do 14.   msg ← shared_mem.msgBuffer<sub>rj</sub>.read ( ); 15.   if msn.type = ACK_NEW_LOCAL_VIEW ∧ (msg ∉ ListACK[ ]) then 16.     if verifySign (msg.Sign, msg.Cert) ∧ (L(f) = msg.L) ∧ (G(f) = msg.G) then 17.       ListACK (msg.id<sub>m</sub>, msg.L, msg.G)            ← ListACK (msg.id<sub>m</sub>, msg.L, msg.G) ∪ {msg}; 18.       if  ListACK (msg.id<sub>m</sub>, msg.L, msg.G)  ≥ g+1 then 19.         if msg.id<sub>m</sub> = δ<sub>prot,2</sub>.id then 20.           δ<sub>prot,2</sub>.time ← ⊥; 21.         end if 22.       end if 23.     end if 24.   end if </pre>
--	---

Na linha 13, o  $AS_j$  é sinalizado sobre presença de mensagem em memória compartilhada. A leitura da mensagem, a verificação da mesma como sendo do tipo  $ACK\_NEW\_LOCAL\_VIEW$  e também a constatação que a mesma é uma mensagem nova, são realizadas nas linhas 14 e 15 deste algoritmo. Na linha subsequente (linha 16), outras verificações são realizadas. A mensagem  $acknewLocal_k$  recebida tem a sua assinatura validada e um filtro é aplicado para evitar o processamento de reconhecimentos antigos e não recebidos por  $S_j$ . Este filtro ( $(L(f) = msg.L) \wedge (G(f) = msg.G)$ ) deve passar somente reconhecimentos que retornem os

valores de visão local ( $L_j(j)$ ) e global ( $G_j(j)$ ) de  $S_j$  que foram enviados previamente aos outros sítios do sistema na mensagem *newLocalView<sub>rj</sub>* correspondente. Uma vez que tenha passado pelos testes citados, a mensagem recebida é incluída na lista apropriada (*ListACK(msg.id<sub>m</sub>, msg.L, msg.G)*) (linha 17).

Outro teste, na sequência (linha 18), verifica se já ocorreram  $g + 1$  ou mais mensagens *acknewLocal<sub>k</sub>*'s na lista *ListACK(msg.id<sub>m</sub>, msg.L, msg.G)*. Uma vez atingido este limite de mensagens, o sítio  $S_j$  assume que sua réplica coordenadora  $R_{rj}$  é conhecida pelo menos por um quórum mínimo necessário de sítios do sistema. Com isto, nas linhas 19 e 20, é desativado o *timeout* ligado à estrutura  $\delta_{prot\_2}$  que foi criada com o prazo *prazoAccept* para a espera destas mensagens de *acknewLocal<sub>k</sub>*. Este prazo e a criação da estrutura correspondente foram estabelecidos anteriormente, nas linhas entre 40 e 42 no algoritmo 6.

#### 5.4.2.2 Tratamento de *timeouts* na Segunda Parte do Protocolo

O procedimento no tratamento de exceções referente a ultrapassagens do prazo *prazoAccept*, apresentado nesta seção, é praticamente o mesmo do apresentado para *prazoCoord* na seção 5.4.1 (algoritmo 8). As diferenças estão no fato que nesta seção tratamos com exceções relacionadas a dois tipos de mensagens: os *accept<sub>k</sub>*'s e os *acknewLocal<sub>k</sub>*'s.

O algoritmo 13 que se segue, apresenta na linha 11 o controle de ultrapassagens do *timeout prazoAccept* que ativa tratamentos de exceções associados com dois tipos diferentes de exceções no nosso esquema. No primeiro caso, são discutidas as exceções ligadas ao *timeout (prazoAccept)* estabelecido através da estrutura  $\delta_{prot\_2}$  de *timeout* no algoritmo 4 (linhas 28 a 30 do algoritmo 4, seção 5.2.5). O intervalo de tempo *prazoAccept* foi estipulado como prazo máximo para a chegada de mensagens *accept<sub>k</sub>* no *Agreement Service* do sítio  $S_j$ . Este *timeout* está associado à situação em que o limite  $g + 1$  de mensagens de *accept<sub>k</sub>* na linha 18 no algoritmo 5 (seção 5.2.6) não foi atingido no prazo definido (*prazoAccept*).

O outro caso corresponde a exceções ligadas ao prazo (*prazoAccept*) estabelecido também através de uma estrutura  $\delta_{prot\_2}$  de *timeout* mas, desta vez, no algoritmo 6 (linhas 40 a 41 do algoritmo 6, seção 5.2.7). O intervalo de tempo correspondente está associado desta vez à chegada de no mínimo  $g + 1$  mensagens de *acknewLocal<sub>k</sub>* no *Agreement Service* do sítio  $S_j$  (algoritmo 12). Estes dois casos são

diferenciados pela variável *sync* disponível na estrutura  $\delta_{prot\_2}$  ( $\delta_{prot\_2.sync}$ ).

Na linha 12 do algoritmo 13, é realizado um teste que separa o tratamento das exceções destes dois tipos de mensagens. O teste verifica se a estrutura de *timeout* é associada a *accept<sub>k</sub>*'s ou a *acknewLocal<sub>k</sub>*'s, tomando como base o valor do campo *sync* da estrutura  $\delta_{prot\_2}$  que teve a ultrapassagem do seu prazo de *timeout*. Se este teste indicar que o tratamento da exceção envolve a troca de visão local complementada com certificados (campo *sync*  $\neq null$ ), são então executadas as linhas entre 14 e 23 do algoritmo desta seção. É o tratamento que envolve perdas de mensagens *accept<sub>k</sub>*.

**Algoritmo 13.** Tratamento de *timeouts* da Segunda Parte do Protocolo

<pre> {Replica R<sub>rj</sub>} 1. upon msgSignal at R<sub>rj</sub> do 2. msg ← shared_mem.protBuffer.read(); 3. if msg.type = NEW_LOCAL_VIEW then 4. coord<sub>rj</sub>[j] ← msg.coord; 5. if id<sub>rj</sub> = coord<sub>rj</sub>[j] then 6. newLocalView<sub>rj</sub> ← msg; 7. newLocalView<sub>rj</sub>.id ← id<sub>rj</sub>; 8. send newLocalView<sub>rj</sub> to all S<sub>i</sub>; 9. end if 10. end if </pre>	<pre> {Agreement Service AS<sub>j</sub>} 11. upon (timer() - δ<sub>prot_2.time</sub>) ≥ prazoAccept at AS<sub>j</sub> do 12. if δ<sub>prot_2.sync</sub> ≠ null then 13. if  accept(Cache<sub>j</sub>(δ<sub>prot_2.id<sub>m</sub></sub>)  &lt; g+1 then 14. if int &lt; f+1 then 15. coord[j] ← new_coord(list_replicas<sub>j</sub>, int); 16. L<sub>j</sub> ← L<sub>j</sub> + 1; 17. Sign<sub>AS<sub>j</sub></sub> ← Signature (&lt;NEW_LOCAL_VIEW, δ<sub>prot_2.id<sub>m</sub></sub>, coord[j], δ<sub>prot_2.sync</sub>, L<sub>j</sub>, G&gt;, D<sub>AS<sub>j</sub></sub>); 18. newLocalView<sub>k</sub> ← &lt;NEW_LOCAL_VIEW, null, δ<sub>prot_2.id<sub>m</sub></sub>, coord[j], δ<sub>prot_2.sync</sub>, L<sub>j</sub>, G, Sign<sub>AS<sub>j</sub></sub>, Cert<sub>AS<sub>j</sub></sub>&gt;; 19. shared_mem.protBuffer.write(newLocalView<sub>k</sub>); 20. end if 21. δ<sub>prot_2.time</sub> ← timer(); 22. prazoAccept ← Estimate (prazoAccept); 23. end if 24. else 25. if  ListACK(δ<sub>view.id<sub>m</sub></sub>, δ<sub>view.L</sub>, δ<sub>view.G</sub>)  &lt; g+1 then 26. if int &lt; f+1 then 27. coord[j] ← new_coord(list_replicas<sub>j</sub>, int); 28. L<sub>j</sub> ← L<sub>j</sub> + 1; 29. Sign<sub>AS<sub>j</sub></sub> ← Signature (&lt;NEW_LOCAL_VIEW, δ<sub>prot_2.id<sub>m</sub></sub>, coord[j], δ<sub>prot_2.sync</sub>, L<sub>j</sub>, G&gt;, D<sub>AS<sub>j</sub></sub>); 30. newLocalView<sub>k</sub> ← &lt;NEW_LOCAL_VIEW, null, δ<sub>prot_2.id<sub>m</sub></sub>, coord[j], δ<sub>prot_2.sync</sub>, L<sub>j</sub>, G, Sign<sub>AS<sub>j</sub></sub>, Cert<sub>AS<sub>j</sub></sub>&gt;; 31. shared_mem.protBuffer.write(newLocalView<sub>k</sub>); 32. end if 33. δ<sub>prot_2.time</sub> ← timer(); 34. prazoAccept ← Estimate (prazoAccept); 35. end if 36. end if </pre>
---	---

No caso citado (prazo para *accept<sub>k</sub>*'s), o teste da linha 13 serve de filtro para o caso da chegada de mensagens de *accept<sub>k</sub>* no quórum necessário, simultaneamente com o disparo do *timeout* correspondente. Este filtro que basicamente verifica se as mensagens *accept<sub>k</sub>* necessárias

já se encontram em cache, evita processamentos desnecessários com trocas para recuperar o sítio  $S_j$ .

Na linha 14, o contador de trocas de visão (*int*) indica se é factível continuar a fazer trocas de visão local para tratar estas exceções ou não. Na condição de ainda ser possível trocas de visão local ( $int < f + 1$ ), o  $AS_j$  suspeita de sua coordenadora local ( $R_{rj}$ ) e prepara uma troca de *view* local através da função  $new\_coord(list\_replicas_j, int)$ . Um novo coordenador é determinado e, como consequência, o número de visão local ( $L_j$ ) é também incrementado (linhas 15-16). Uma mensagem  $newLocalView_k$  é então preparada nas linhas 17 e 18, comportando as informações já citadas antes ( $id_m$ ,  $coord_j$ ,  $sync$ ,  $L_j$  e  $G$ ) no algoritmo 8 (seção 5.4.1). Esta mensagem é enviada via memória compartilhada para as réplicas  $R_{rj}$  de  $S_j$  (linha 19 do algoritmo 13).

É importante citar que a condição de  $int > f + 1$  assumida no algoritmo 8 (seção 5.4.1) não teria sentido aqui nesta seção. A condição  $int > f + 1$  no algoritmo 8 associava o comportamento malicioso ao sítio líder e, na sequência, era montada uma mensagem de troca de visão global ( $gviewChange_k$ ). No caso presente (algoritmo 13), consideramos que as trocas de mensagens *accept*'s são generalizadas e não centradas sob o sítio líder, como no caso da mensagem de *propose* tratada no algoritmo 8. Portanto, pode ser totalmente sem sentido ficar trocando de sítio líder quando ocorrem problemas na recepção de mensagens *accept*<sub>k</sub>.

Então, na condição de  $int \geq f + 1$ , não são provocadas trocas de visão global (e envios de mensagens  $gviewChange_k$ ) no algoritmo 13. Com o valor de *int* igual ou superando a  $f + 1$ , em termos de trocas de visão local, representa que pelo menos um coordenador correto representou  $S_j$  no *Paxos*. Neste caso, no algoritmo 13, o problema é então assumido como sendo da rede, devido ao congestionamento da mesma e, como consequência, mensagens de troca de visão (local ou global) não são enviadas.

Ações que são efetivadas para qualquer valor do contador *int* e que ajudam a melhorar as condições de recepção em rede congestionada, são realizadas nas linhas 21 e 22. A estrutura de *timeout*  $\delta_{prot\_2}$  (linha 21) é reativada com a função *timer*() e o novo prazo (*prazoAccept*) é estimado na condição de dar mais tempo para que as mensagens *accept*<sub>k</sub> dos  $2g + 1$  sítios do sistema cheguem em  $S_j$  (linha 22).

A condição ( $\delta_{prot\_2}.sync = null$ ) do teste da linha 12 provoca a execução do processamento entre as linhas 24 e 38. Esta condição corresponde ao tratamento de perdas de mensagens *acknewLocal*<sub>k</sub>. O procedimento que trata estas exceções envolve passos idênticos aos que

descrevemos acima para as perdas de mensagens de  $accept_k$ . Poucas diferenças são notadas. A primeira se refere às listas usadas nos testes das linhas 13 e 25. Outro aspecto a se considerar nas diferenças é que tanto na estrutura de  $timeout_{\delta_{prot\_2}}$  como a mensagem gerada de  $newLocalView_k$  possuem os campos de  $sync$  com valores diferentes. No tratamento de mensagens  $accept_k$ , a variável  $sync$  ( $sync \neq null$ ) indica que haverá trocas de certificados para a atualização das réplicas associadas à mensagem gerada de  $newLocalView_k$ . Já no caso tratamento de perdas de mensagens  $acknewLocal_k$ , a variável  $sync$  ( $sync = null$ ) indica que não haverá trocas de certificados para associadas à mensagem gerada de  $newLocalView_k$ .

Nas linhas entre 1 e 3, as réplicas  $R_{rj}$  recebem mensagem do tipo  $NEW\_LOCAL\_VIEW$  enviada pelo  $AS_j$  via memória compartilhada. O novo coordenador de  $S_j$  é registrado pela réplica  $R_{rj}$  (em  $coord_{rj}[j]$ ). A réplica coordenadora de  $S_j$  então executa as ações entre as linhas 5 e 8 que é o mesmo procedimento do algoritmo 8 (seção 5.4.1). Somente a nova coordenadora envia para os outros sítios a mensagem  $newLocalView_{rj}$ .

As mensagens  $newLocalView_{rj}$  geradas e emitidas pelo algoritmo 13 passam pelos mesmos processamentos apresentados no algoritmo 9 (seção 5.4.1.1).

## 5.5 OPERAÇÕES DE INICIAÇÃO E REMOÇÃO DE RÉPLICAS

As adições e remoções de réplicas com as respectivas máquinas virtuais ( $VMs$ ) são usadas nos algoritmos apresentados anteriormente nas reconfigurações do esquema, passando da configuração mínima ( $f + 1$  réplicas) para a máxima ( $2f + 1$  réplicas) e vice-versa. A configuração máxima só é implementada quando o acordo não é alcançado com as réplicas de um sítio nos resultados de execução de uma operação requisitada por um cliente. Depois de conseguido este acordo na configuração máxima e a mensagem de  $reply$  já ter sido enviada ao cliente com os resultados acordados, a configuração deve voltar a configuração mínima, excluindo as réplicas maliciosas e mesmo as corretas que excedem o número de  $f + 1$  réplicas (que é a configuração mínima). Duas operações foram desenvolvidas no nosso sistema para atender estas passagens de configuração que são as funções  $initiateVMs()$  e  $restoreVMs()$ .

### 5.5.1 Função de Iniciação de VMs

O algoritmo 14 mostra a função *initiateVMs()* que é executada em um *Agreement Service* e que permite a iniciação de  $f$  novas VMs, cada uma destas com uma nova réplica em um sítio  $S_j$  de nosso esquema (linha 1).

Esta função apresenta como um de seus parâmetros o  $f$  que representa o número de réplicas (e suas VMs) que devem ser criadas conforme as necessidades da passagem para a configuração máxima. Este número é armazenado na variável *numReplica* na linha 2.

Algoritmo 14: Função de iniciação de VMs

<pre> {Agreement Service AS}  1. function initiateVMs(<math>f, id_m</math>) 2.   numReplica <math>\leftarrow f</math>; 3.   gOrder<sub>m</sub> <math>\leftarrow</math> orderList.getOrder(<math>id_m</math>); 4.   hash<sub>s</sub> <math>\leftarrow</math> stableMemory_Hash.read(); 5.   for <math>\forall r : numReplica + 2 \leq r \leq numReplica + 1</math> do 6.     msgStart<sub>r</sub> <math>\leftarrow</math> &lt;MSG_START, null, <math>id_m</math>, gOrder<sub>m</sub>, hash<sub>s</sub>&gt;; 7.     createContextVM(<math>R_r</math>); 8.     startVM(<math>R_r</math>); 9.     shared_mem.protBuffer.write(msgStart<sub>r</sub>); 10.    list_replicas<sub>r</sub> <math>\leftarrow</math> &lt;<math>R_r</math>.id, <math>R_r</math>.networkAddress&gt; 11.  end for 12.  return list_replicas<sub>r</sub> 13. end function </pre>	<pre> {Réplica <math>R_j</math>}  Init: 14. state<sub>j</sub> <math>\leftarrow</math> 0; 15. temp_state <math>\leftarrow</math> 0; 16. newReplica <math>\leftarrow</math> true;  17. upon msgSignal at <math>R_j</math> do 18.   msg <math>\leftarrow</math> shared_mem.protBuffer.read(); 19.   if msg.type = MSG_START <math>\wedge</math> newReplica = true then 20.     hash<sub>s</sub> <math>\leftarrow</math> msg.hash; 21.     id<sub>m</sub> <math>\leftarrow</math> msg.id<sub>m</sub>; 22.     gOrder<sub>m</sub> <math>\leftarrow</math> msg.gOrder; 23.     s <math>\leftarrow</math> 0; 24.     while (state<sub>j</sub> = 0) do 25.       s <math>\leftarrow</math> s+1; 26.       temp_state <math>\leftarrow</math> stableMemory_Checkpoint<sub>j</sub>.read(hash<sub>s</sub>.number); 27.       calcHash<sub>j</sub> <math>\leftarrow</math> calculateHash(temp_state); 28.       if calcHash<sub>j</sub> = hash<sub>s</sub>.resume then 29.         state<sub>j</sub> <math>\leftarrow</math> temp_state; 30.       end if 31.     end while 32.     replicaControl<sub>j</sub> <math>\leftarrow</math> state<sub>j</sub>.replicaControl; 33.     nextExecution<sub>j</sub> <math>\leftarrow</math> replicaControl<sub>j</sub>.nextExecution; 34.     checkpointNumber<sub>j</sub> <math>\leftarrow</math> hash<sub>s</sub>.nextCheckpoint; 35.     for <math>\forall msg_s \in</math> shared_mem.buffer_toExecute : 36.       (msg<sub>s</sub>.gOrder = nextExecution<sub>j</sub>) <math>\wedge</math> (msg<sub>s</sub>.gOrder <math>\leq</math> gOrder<sub>m</sub>) do 37.         data<sub>s</sub> <math>\leftarrow</math> shared_mem.buffer_toExecute.read(msg<sub>s</sub>.gOrder); 38.         req<sub>s</sub> <math>\leftarrow</math> data<sub>s</sub>.req; 39.         resp <math>\leftarrow</math> appThread.execute(req<sub>s</sub>.op); 40.         if msg<sub>s</sub>.gOrder = gOrder<sub>m</sub> then 41.           response<sub>j</sub> <math>\leftarrow</math> &lt;RESPONSE, <math>R_j</math>.id, req<sub>s</sub>.id<sub>m</sub>, resp&gt;; 42.           if data<sub>s</sub>.gOrder = checkpointNumber<sub>j</sub> then 43.             checkpoint(checkpointNumber<sub>j</sub>); 44.             checkpointNumber<sub>j</sub> <math>\leftarrow</math> checkpointNumber<sub>j</sub> + nextPoint; 45.           end if 46.           shared_mem.appBuffer<sub>j</sub>.write(response<sub>j</sub>); 47.         end if 48.       nextExecution<sub>j</sub> <math>\leftarrow</math> nextExecution<sub>j</sub> + 1; 49.     end for 50.     newReplica <math>\leftarrow</math> false; 51.   end if </pre>
---	---

O segundo parâmetro desta função é a identificação da última requisição de cliente ( $id_m$ ) que o estado destas novas réplicas devem

apresentar registros de execução antes de retornarem como réplicas ativas do esquema. O identificador  $id_m$  (ou identificador da instância do protocolo de acordo distribuído) é usado para buscar a ordem global atribuída à requisição correspondente ( $gOrder_m$ , linha 3). Também nesta função ocorre a busca da estrutura  $hash_k$  correspondente ao último *checkpoint* realizado (linha 4).

Como esta função *initiateVMs()* só é executada quando não existe acordo em uma configuração mínima (com  $f + 1$  réplicas) e, para se conseguir este mesmo acordo em presença de até  $f$  réplicas maliciosas precisamos criar mais  $f$  réplicas. A passagem para a configuração máxima ( $2f + 1$ ) permite então que o acordo necessário seja alcançado. Esta criação das  $f$  VMs/réplicas adicionais é concretizada entre as linhas 5 e 10 do algoritmo 14. As novas réplicas  $R_{rj}$  são identificadas com  $r$  variando de  $f + 2$  até  $2f + 1$  no *loop* das linhas indicadas.

Para cada réplica  $R_{rj}$  criada, é montada uma estrutura  $msgStar_{rj}$  na linha 6 que possui como campos: a identificação da última requisição em execução no sistema ( $id_m$ ), a ordem global recebida por esta requisição ( $gOrder_m$ ) e o  $hash_k$  correspondente ao último *checkpoint* feito no sistema. Na linha 7, a função *createContextVM()* inicializa *buffers* em memória compartilhada para  $R_{rj}$ . Depois, na linha 8, a função *startVM()* dá partida na VM da réplica  $R_{rj}$ . Estas duas últimas funções usam interface do suporte de virtualização disponível. A mensagem  $msgStar_{rj}$  é então enviada via memória compartilhada pelo  $AS_j$  para a réplica  $R_{rj}$  recém criada (linha 9). Por fim, na linha 10 a lista com as réplicas de  $S_j$  ( $list\_replicas_j$ ) é atualizada com as informações da réplica criada ( $R_{rj}.id$ ,  $R_{rj}.networkAdress$ ). Na linha 12 a função *initiateVMs()* retorna  $list\_replicas_j$ .

Réplicas recém-iniciadas são as únicas capazes de executar mensagens  $msgStar_{rj}$ . Na sinalização da linha 17, a Réplica  $R_{rj}$  lê a mensagem  $MSG\_START$  (linha 18). Verifica no teste o tipo da mensagem e se é réplica nova (linha 19). Na sequência de linhas 20, 21 e 22, são recuperados desta mensagem de *start* a estrutura  $hash_k$  do último *checkpoint*, o identificador ( $id_m$ ) da última requisição executada no sistema e a ordem global  $gOrder_m$  desta requisição. O estado da réplica  $R_{rj}$  é assumido de início como vazio ( $state = \emptyset$ , linha 14).

Entre as linhas 24 e 31, é executado um comando *while* que controla as várias iterações usadas para a busca de um estado correto da memória estável. Na linha 24 é verificada a condição que vai permitir então esta busca de uma cópia de estado ( $replicaState_{sj}$ ) correto de uma réplica ( $R_{sj}$ ) já ativa. Para a busca de um estado, na linha 26, é usado o número do *checkpoint* disponível na estrutura  $hash_k$  ( $hash_k.number$ ). Este



estado  $replicaState_{sj}$  é copiado em variável temporária ( $temp\_state$ ) (linha 27). Depois, na linha 28, é calculado o resumo deste estado recuperado ( $calcHash_{sj}$ ). Se este resumo calculado coincide com o resumo obtido da estrutura  $hash_k$ , o estado recuperado ( $replicaState_{sj}$ ) da réplica ativa é então correto. Como consequência, este estado mantido como temporário ( $temp\_state$ ), é então efetivado como estado da nova réplica (linha 29). Se houver falha nesta comparação de  $hashes$ , o contador  $s$  (linha 25) é incrementado e o estado de outra réplica  $R_{sj}$  é obtido na memória estável para o mesmo teste.

Nas linhas 32 e 33, é recuperado do estado da réplica, primeiro o controle de execução ( $replicaControl_{rj}$  na linha 32) que permite, na sequência, a obtenção do ponteiro de próxima requisição a ser executada ( $nextExecution_{rj}$  na linha 33). O número do próximo *checkpoint* ( $checkpointNumber_{rj}$ ) é recuperado na linha 34 a partir da estrutura  $hash_k$  ( $hash_k.nextCheckpoint$ ).

Com estas informações recuperadas, a réplica  $R_{rj}$  pode buscar as requisições (mensagens) no *buffer shared mem.buffer toExecute* que foram admitidas no sistema desde o último *checkpoint* e até a última requisição já executada por réplicas ativas (linhas 35 e 48). As requisições são recuperadas na ordem global, controladas pelo ponteiro  $nextExecution_{rj}$  e executadas nesta mesma ordem na linha 38. Estas execuções são interrompidas quando a ordem global da requisição a ser executada ultrapassa o valor da última execução das réplicas ativas do sistema ( $msg_k.gOrder = gOrder_m$ ), na linha 39).

Na sequência, a réplica  $R_{rj}$  prepara a mensagem  $response_{rj}$  com os resultados desta última execução para enviar para o  $AS_j$  (linha 40). Se a ordem global da requisição executada corresponde ao próximo *checkpoint*, então as operações correspondentes são executadas entre as linhas 41 e 44. Nesta situação, a réplica recuperou o estado referente a instância de protocolo de identificador  $id_m$ . A réplica  $R_{rj}$  então envia via memória a mensagem  $response_{rj}$  para o  $AS_j$  (linha 45). Na linha 47, o ponteiro  $nextExecution_{rj}$  é incrementado para o controle de execuções no comando *for* definido (entre as linhas 35 e 48). Uma vez atualizado o seu estado com informações da instância de protocolo de  $id_m$ , a réplica  $R_{rj}$  se registra como ativa e deixa de ser réplica nova (linha 49), finalizando assim a sua iniciação através do algoritmo 14.

### 5.5.2 Função de Remoção de VMs

A função *restore()* é usada para restaurar a configuração mínima quando a replicação Máquina de Estado (ME) termina a segunda fase do protocolo de acordo em configuração máxima ( $2f + 1$  réplicas). O

algoritmo 15 mostra as ações que concretizam esta passagem para a configuração mínima ( $f+1$  réplicas).

Esta função é chamada com passagem dos parâmetros  $f+1$  e  $id_m$ . O primeiro define o número de réplicas que deve ficar no final desta restauração aplicada na máquina de estados do sítio  $S_j$ . O identificador  $id_m$  determina a instância de protocolo que está sendo concluída.

**Algoritmo 15.** Função de remoção de *VMs*

{*Agreement Service AS<sub>j</sub>*}

```

1. function restoreVMs ( $f+1, id_m$ )
2.   for  $\forall r : R_{rj} \in list\_replicas_j$  do
3.     appThreadrj.stop();
4.   end for
5.   while  $\forall r, \exists (response_{rj}) \in respList(id_m)$  do
6.     responsek ← respList(idm).read();
7.     replyk ← replyCachej.read(idm);
8.     if responsek.resp ≠ replyk.resp then
9.       stopVM(Rrj);
10.      clearContextVM(Rrj);
11.      list_replicasj ← list_replicasj \ Rrj;
12.    end if
13.  end while
14.  while  $|list\_replicas_j| > f+1$  do
15.     $n \leftarrow |list\_replicas_j|$ ;
16.    exceedReplica ← order(list_replicasj, n);
17.    stopVM(exceedReplica);
18.    clearContextVM(exceedReplica);
19.    list_replicasj ← list_replicasj \ exceedReplica;
20.  end while
21.  for  $\forall r : R_{rj} \in list\_replicas_j$  do
22.    appThreadrj.resume();
23.  end for
24.  return list_replicasj;
25. end function

```

O algoritmo 15 definido na função *restore()*, a nível de *Agreement Service*, começa com o comando *for* entre as linhas 2 e 4, cuja finalidade é parar todas as *threads* que executam nas diferentes réplicas da ME. Logo depois, nas linhas entre 5 e 12, é percorrida a lista *respList(id<sub>m</sub>)*, recuperando as mensagens *response<sub>rj</sub>*, uma a uma, de cada réplica  $R_{rj}$  (linha 6). Estas mensagens possuem os resultados de processamento de cada réplica no campo *resp* de cada uma destas mensagens referentes a *req<sub>k</sub>* do cliente. Na sequência, a mensagem *reply<sub>k</sub>* é recuperada em cache (linha 7). Esta mensagem *reply<sub>k</sub>* possui o resultado acordado para este processamento. Ou seja, o valor expresso pela variável *reply<sub>k</sub>.resp* é o resultado acordado e enviado ao cliente como correto para o processamento da operação definida por *req<sub>k</sub>*. Este valor em *reply<sub>k</sub>.resp* é usado para detectar as réplicas que apresentam comportamento incorreto (ou malicioso).

Na linha 8, um teste compara os resultados das mensagens *response<sub>k</sub>*, uma a uma, com o resultado correto de processamento da mensagem *reply<sub>k</sub>*. As réplicas  $R_{rj}$  que tiverem seus resultados (*response<sub>k</sub>.resp*) divergente da mensagem *reply<sub>k</sub>*, devem ter primeiro suas *VMs* sofrendo os efeitos da operação de parada (*stopVM*) na linha 9. Depois, na sequência, o contexto da réplica é removido

(*clearContextVM(R<sub>ij</sub>)*) (linha 10) e a réplica tem suas informações excluídas da lista de réplicas do sítio  $S_j$  (*list\_replicas<sub>j</sub> \ R<sub>ij</sub>*) (linha 11).

Os passos executados até aqui retiraram as réplicas maliciosas que diferiram do resultado acordado. Mas o número de réplicas retiradas da ME podem ainda não ter levado esta replicação para uma configuração mínima. Ou seja, podemos ainda ter réplicas corretas em número maior que o limite de uma configuração mínima ( $f+1$  réplicas).

Entre as linhas 14 e 20, é executada uma estratégia que retira réplicas ainda corretas para deixar a ME em configuração mínima. As iterações do comando *while* se executam até que esta configuração mínima seja alcançada. Na linha 15, é atribuído à variável  $n$  o valor correspondente ao número de réplicas na configuração da ME.

O valor de  $n$  permite determinar na linha 16, com a função *order*, a réplica de maior *id* na lista de réplicas (*list\_replicas<sub>j</sub>*), ou seja, indica uma réplica recente que foi criada pela função *initiateVM()* (seção 5.5.1). Na estratégia usada, esta réplica deve ser retirada da lista de réplicas. As operações entre as linhas 17 e 19 excluem esta réplica da configuração da ME. As operações incluídas no comando *while* (linhas 14-20) são executadas até atingir a configuração mínima, favorecendo sempre as réplicas mais antigas, porque os endereços das réplicas em *list\_replicas<sub>j</sub>* são percorridos em ordem decrescente.

As réplicas que continuam na configuração mínima são reativadas pelo comando *for* entre as linhas 21 e 22. A nova lista de réplicas (*list\_replicas<sub>j</sub>*) é retornada na linha 24 ao contexto chamador ( $AS_j$ ).

A aplicação da função *restore()* pode excluir o coordenador que concluiu a instância de protocolo  $id_m$  no algoritmo 6. E isto implica na troca de visão local obrigatória que no caso não necessita de envios de certificados (conforme já foi exposto na seção 5.4.2.1).

## 5.6 CONCLUSÃO DO CAPÍTULO

Neste capítulo, introduzimos a extensão do modelo ITVM tinha sido apresentado no capítulo anterior. Este modelo estendido foi introduzido, primeiro caracterizando seu modelo de sistema onde as premissas foram aproximadas de ambientes reais. Uma indicação disto foi a adoção de *fair links* para canais de comunicação. Nossos canais, a partir deste modelo admitem a perda de mensagens.

Depois do modelo, apresentamos as entidades que atuam nos dois níveis de sistema que definimos. O modelo apresenta as réplicas da ME agrupadas em sítios e dois níveis de acordo são usados. O primeiro a nível local, baseados em técnicas de BFT, fica muito simplificado pelo uso de componente confiável. O segundo nível, devido as técnicas que

restringem a ação de réplicas maliciosas, o comportamento destas em nível global pode facilmente ser modelado como faltas de *crash*. Neste sentido usamos o algoritmo já clássico do *Paxos*.

O Protocolo hierárquico, apresentado neste capítulo em detalhes, deve servir de base para serviços tolerantes a intrusões e faltas em redes *WANs*. No próximo capítulo, apresentamos a análise do protocolo descrito e um protótipo do mesmo desenvolvido com o objetivo de testar as características do mesmo.

## 6 RESULTADOS EXPERIMENTAIS

O presente capítulo apresenta análises realizadas sobre o protocolo hierárquico proposto para o nosso esquema de replicação ME. Primeiro analisamos a nossa proposta à luz de propriedades desejáveis em sistemas distribuídos assíncronos. Depois na sequência procuramos evidenciar que mesmo em condições de elementos faltosos ou corrompidos as instâncias de protocolo evoluem e conseguem terminar. Na sequência, descrevemos os nossos resultados de protótipo e testes desenvolvidos para verificar o comportamento do protocolo proposto. Por fim são discutidos os resultados conseguidos confrontando com a literatura relacionada.

### 6.1 ANÁLISE DO SUPORTE ALGORÍTMICO HIERÁRQUICO

A manutenção de propriedades é importante em qualquer algoritmo distribuído no sentido de atestar sua correção. A propriedade de *safety* deve sempre ser mantida em qualquer protocolo, mas a vivacidade só é satisfeita em *rounds* favoráveis (no nosso algoritmo, visões globais favoráveis). Um *round* é considerado favorável quando seu sítio líder é correto e o sistema está num período de sincronismo: as comunicações e computações ocorrem dentro de um período de tempo limitado. Nesta situação, um valor proposto pode ser aprendido (decidido) dentro do período de um *round*. Caso um *round* não seja favorável, um novo *round* é iniciado com um novo líder e assim sucessivamente. Se estivermos atravessando um período de sincronismo um valor pode ser aprendido em até  $f+1$  *rounds*.

#### 6.1.1 Propriedades do *Paxos*

O nosso modelo, se analisado em termos de trocas de mensagens entre sítios, corresponde ao *Paxos* e poderíamos nos fixar nas propriedades deste protocolo que são extensamente provadas no artigo original [Lamport, 1998] [Lamport, 2001]. Mas levamos em consideração que os nossos componentes podem ser maliciosos e que, pelo fato de estarem devidamente contidos, apresentam um comportamento no máximo de falhas de *crash*. Diante disto, repassamos as propriedades do *Paxos*, tentando verificar se as condições de segurança previstas realmente amarram os componentes a situação de falhas benignas, facilmente contornáveis.

As propriedades que o *Paxos* deve manter são que:

*Safety*: Se dois servidores (no nosso caso sítios) executam um *update*, então este *update* é o mesmo.

O nosso algoritmo que é um *Paxos* modificado para atender a hierarquia no acordo também mantém esta propriedade. Temporizações e a condição da linha 22 do algoritmo 4 (seção 5.2.5) afastam qualquer participação do sítio líder para trazer inconsistências aos números de ordem global atribuídos as requisições de clientes. Outras situações também ligas ao *safety* são discutidas abaixo. Portanto, o nosso esquema, acreditamos, deve manter esta propriedade.

Validade: *Somente um update introduzido por um cliente pode ser executado.*

No nosso esquema existe a troca de assinaturas e de certificados. Somente um cliente identificado pode executar uma operação na ME do serviço replicado. A assinatura do cliente é verificada antes da admissão da requisição do cliente e na recepção da mensagem de admissão ( $reqS_k$ ). Portanto, fica difícil imaginar qualquer corrupção no sistema e o não atendimento da validade.

At most Once: *Se um servidor executa um update usando uma numeração sequencial. Então o mesmo update executado com um número de sequência  $i$ , não executará sob outro número  $i'$ , tal que  $i \neq i'$ .*

Esta situação é também controlada no nosso sistema que convive com componentes maliciosos. As execuções nas réplicas seguem números globais de sequência. Números de sequência só são atribuídos pelo sítio líder, através de seu *AS* que no nosso sistema é considerado um elemento confiável. Réplicas executando as requisições não podem alterar a ordem de execução, porque recebem as mesmas, uma a uma, em ordem da sequência global de seus *AS*'s que são também elementos confiáveis.

Localmente, dentro dos sítios, as condições de *safety* já foram discutidas no capítulo anterior.

Para a terminação do algoritmo, é necessário que condições favoráveis ocorram nas comunicações entre cliente e servidores. Como assumimos e comportamento *parcialmente síncrono* tanto na rede como na memória compartilhada, as nossas execuções de protocolo devem atravessar períodos de sincronismo e terminarem. No modelo de sistema distribuído deste capítulo assumimos o comportamento de *fair links* onde mensagens podem ser perdidas. As temporizações armadas devem provocar possíveis recuperações com retransmissões das réplicas e dos clientes (ligadas às retransmissões de requisições e *replies*). As condições de *fair link* garantem que com as retransmissões, mesmo assim, as mensagens acabariam por chegar nas réplicas (ou no cliente se for o caso). Portanto, sabendo que o *Paxos* funciona com estes canais, em ambiente parcialmente síncrono e que os limites de faltas adotados não são

ultrapassados, um cliente então sempre retornará para sua aplicação com uma resposta em tempo finito, assegurando a terminação do acordo.

### 6.1.2 Recuperação do Cliente em Instâncias de Protocolo

Se olharmos o algoritmo do cliente (algoritmo 1), basta que pelo menos uma mensagem correta de  $reply_k$  atinja o cliente. Se houver a expiração do *timeout* do cliente (linha 20 do algoritmo 1), o procedimento tomado será enviar cópias de  $req_k$  a uma próxima coordenadora em  $coord_h$  (este *array* foi obtido no repositório de nomes do domínio do sítio  $S_h$ ). Os endereços de  $coord_h$  são em número dos sítios do sistema ( $n \geq 2g + 1$ ) e o cliente deve percorrer esta lista até que receba a resposta.

O *Paxos* apresenta um procedimento não muito diferente deste, uma vez que o cliente só deve receber o *reply* do servidor que recebeu originalmente a requisição do cliente. Como este servidor pode falhar, o cliente do *Paxos* deve recorrer a um controle de temporização.

No nosso sistema o que pode complicar é que as trocas de visões e reconfigurações podem tornar a lista em  $coord_h$  desatualizada (isto não ocorre no *Paxos* porque não existe o tratamento de elementos faltosos). Podemos amenizar o problema fazendo atualizações periódicas dos registros dos  $coord_h$ 's em repositórios de nomes. Mas não podemos admitir que estas frequências sejam altas de modo a interferir no desempenho das instâncias do protocolo. É provável então que clientes convivam com alguma inconsistência nestas listas de coordenadores.

Nas retransmissões de cliente em canais *fair links*, o *timeout* é sempre melhorado (pela função *Estimate()*) com incrementos de  $\Delta$  unidades de tempo, até que consiga atravessar um período de sincronismo no ambiente. Estes reenvios são apresentados nas linhas 20 e 23 do algoritmo 1 (seção 5.2.2). Se a requisição foi executada e, por uma condição momentânea da rede, o  $reply_k$  correspondente não atingir o cliente, reenvios subsequentes de  $req_k$  terão como resposta cópias da  $reply_k$  correspondente que está em *cache* (linhas 21- 23 do algoritmo 2, seção 5.2.3). O comportamento de *fair links* nos enlaces da rede é admitido no modelo do próprio *Paxos*.

Mesmo com canais *fair links*, réplicas poderiam se negar a enviar ao cliente suas respostas autenticadas pelo *Agreement Service* ( $reply_k$ ). A redundância da configuração inicial ( $f + 1$ ) já é suficiente para que pelo menos uma mensagem correta possa atingir o cliente em condições favoráveis de sincronismo. Na nossa abordagem as  $f + 1$  réplicas do sítio que recebeu originalmente a requisição do cliente, enviam a mensagem de  $reply_k$ .

### 6.1.3 Evolução do Protocolo em Situações de Exceções

O nosso esquema hierárquico possui dois controles de temporização armados nas instâncias de protocolo tal que, uma vez não verificada os prazos correspondentes tratadores são ativados. Na sequência examinamos as possibilidades de exceções e suas consequências na evolução do protocolo.

#### 6.1.3.1 Perda de mensagem *propose*:

A situação ilustrada na Figura 6.1 mostra o sítio 2 não recebendo uma mensagem *propose* referente a uma requisição ( $req_k$ ) já admitida no sistema (através de  $reqS_k$ ). No algoritmo 3 (seção 5.2.4), nas linhas 26-28, uma vez a mensagem de admissão correspondente tenha sido recebido o *timeout* com prazo *prazoCoord* é estabelecido para a recepção da mensagem correspondente de *propose*. Se a mensagem chegar no algoritmo 4 (seção 5.2.5), este *timeout* é desativado (linha 22). Mas, no caso da Figura 6.1, onde a mensagem não chegando no prazo, o tratador correspondente nas linhas 19-40 no algoritmo 8 é ativado. Na figura, é indicada uma primeira exceção, então o *AS* do sítio  $S_2$  desconfia de sua réplica coordenadora e faz a troca de visão local, mandando no passo seguinte mensagens de *newLocalView<sub>r,2</sub>*, através de seu novo coordenador  $R_{r,2}$ . Na sequência, troca de certificados são usadas para sincronizar novamente o sistema. Notem que se o sítio líder ( $S_i$ ) continuar mandando *propose*'s somente para uma maioria de sítios, o protocolo continua evoluindo normalmente, mas ao custo de trocas locais de visões locais dos que não recebem esta mensagem do líder. Na verdade basta que esta mensagem atinja um sítio do sistema. E mesmo nesta situação o protocolo evolui com a troca de certificados.

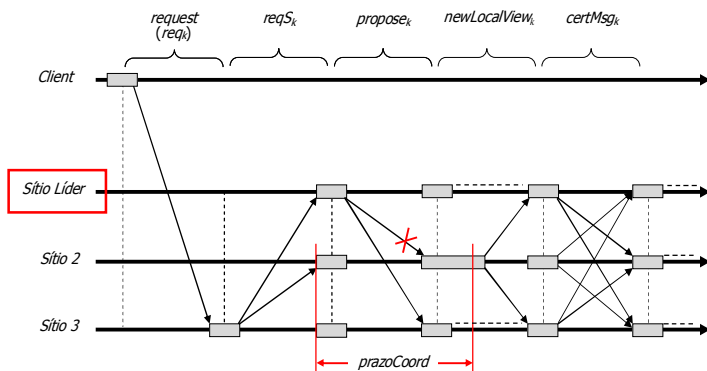


Figura 6.1 – Perda de mensagem *propose*.



### 6.1.3.2 Perda completa de mensagens *propose*:

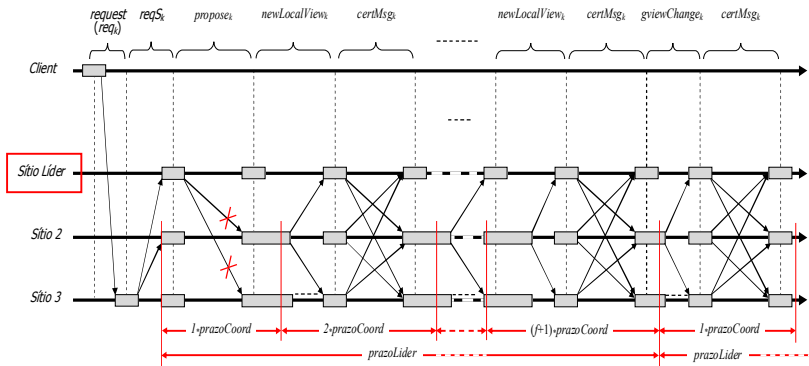


Figura 6.2 – Perda completa de mensagens *propose*.

Se o líder não envia nenhuma mensagem *propose*, como é o caso da Figura 6.2. Nesta situação, os sítios começam a desconfiar de suas coordenadoras, enviam suas mensagens de troca de visão (*newLocalView<sub>r,2</sub>*) e na sequência trocam certificados. Na Figura 6.2, é mostrado somente um sítio enviando mensagem de troca de visão para não dificultar o entendimento da figura. Mas, tanto  $S_2$  como  $S_3$  nesta figura estão trocando de visão.

Como a primeira troca de certificados não resolve o problema. Só o sítio líder possui a mensagem em seu *AS*, mas coordenadora do sítio líder está corrompida. Então os outros sítios não recebem a mensagem *propose*. Com isto *prazoCoord* que foi armado novamente com uma nova estimativa de valor volta a ser ultrapassado e as trocas locais de coordenadoras se sucedem nos outros sítios até que o valor de *int* tenha atingido o valor de  $f + 1$ . Neste caso o prazo do líder foi ultrapassado e ocorre envio de mensagens e certificados (*gviewChange<sub>k</sub>* e *certMsg<sub>k</sub>*) indicando a troca de visão global.

### 6.1.3.3 Perda da mensagem de admissão:

A Figura 6.3 ilustra a situação onde o sítio  $S_2$  não recebeu uma mensagem de admissão (*reqS<sub>k</sub>*) e, por consequência, não processa a mensagem de *propose<sub>k</sub>* correspondente (veja a linha 16 do algoritmo 4, seção 5.2.5). Nesta situação, o sítio  $S_2$  não arma o *timeout prazoCoord* no momento devido. Com a recepção de  $g + 1$  mensagens de *accept<sub>k</sub>* dos outros sítios, o sítio desatualizado (sítio 2) envia mensagem de troca de visão (*newLocalView<sub>2</sub>*), armando *timeout* com *prazoCoord* (linhas 44-56

no algoritmo 5, seção 5.2.6) para receber os certificados necessários para a sua atualização.

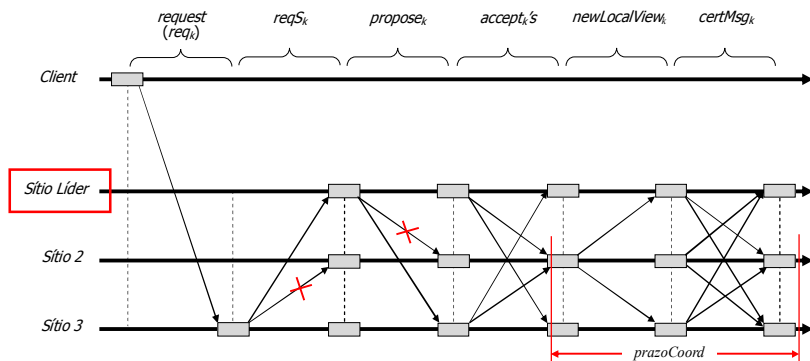


Figura 6.3 – Perda da mensagem de admissão.

#### 6.1.3.4 Não recepção de $accept_k$ 's no quórum necessário:

A Figura 6.4 apresenta a situação em que o sítio  $S_2$  não atingindo o quórum mínimo de mensagens  $accept_k$ 's para que possa evoluir. Ou seja,  $S_2$  não consegue verificar a condição da recepção de no mínimo  $g + 1$  mensagens de  $accept_k$  (linha 29 do algoritmo 5). Nesta situação, não consegue evoluir e o  $prazoAccept$  acaba por ser ultrapassado, gerando uma exceção que ativa o código do tratador correspondente (algoritmo 13).

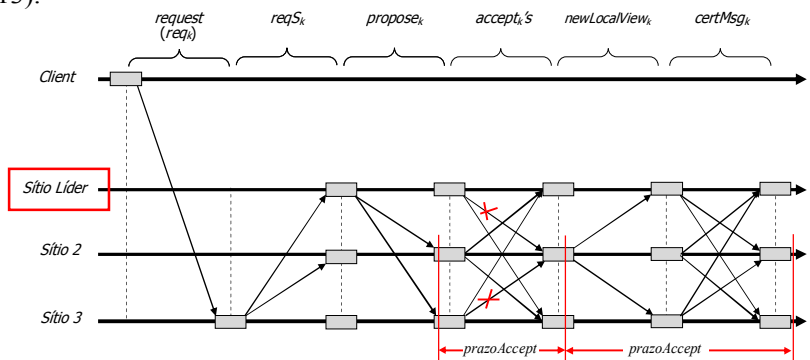


Figura 6.4 – Sítio com recepções insuficientes de mensagens  $accept_k$ 's.

O tratador ativado provoca a troca do líder local com o envio das mensagens correspondentes para sinalizar esta troca e se sincronizar aos outros sítios (trocas de  $newLocalView_2$  e  $certMsg_k$ ). O *timeout*

*prazoAccept* tem novo valor estimado. E se no final da troca de certificados,  $S_2$  ainda não obteve este quórum mínimo, nova troca de visão local ocorre em  $S_2$  com as respectivas trocas de mensagens necessárias (trocas novamente de *newLocalView<sub>2</sub>* e *certMsg<sub>k</sub>*) para a sincronização de  $S_2$ . O processo se repete até  $int = f + 1$ , mas sem a troca de sítio líder.

### 6.1.3.5 Reconfiguração com exclusão de réplica coordenadora:

Com a conclusão da instância de protocolo  $id_m$  no sítio 2, pode ocorrer a restauração provocando a passagem de uma configuração máxima ( $2f + 1$  réplicas) para mínima ( $f + 1$  réplicas) em  $S_2$ . O coordenador pode ser excluído da nova lista *list\_replicas<sub>2</sub>* o que então exige a troca de visão local, definindo um novo coordenador. A Figura 6.5 ilustra esta situação. As mensagens que *newLocalView<sub>2</sub>* são usadas para sinalizar aos outros sítios a troca de coordenador e indicam também a não necessidade de troca de certificados porque  $id_m$  foi concluída. O emissor que sofreu a restauração arma o *timeout prazoAccept* para receber os reconhecimentos dos outros sítios (*acknewLocal<sub>k</sub>*) para as mensagens enviadas sobre troca de visão.

Esta troca de mensagens tem como objetivo de fazer o novo coordenador de  $S_2$  ser conhecido no sistema. As trocas concluem com o sítio  $S_2$  recebendo dos outros sítios  $g + 1$  reconhecimentos (*acknewLocal<sub>k</sub>*).

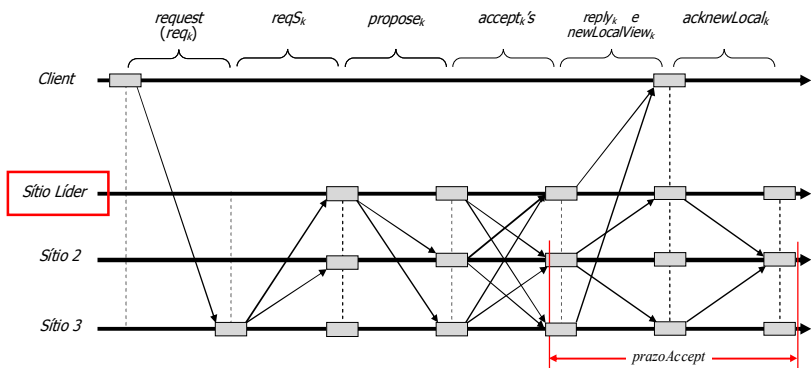


Figura 6.5 – Restauração de configuração mínima com exclusão de coordenador.

### 6.1.4 Custos do Protocolo Proposto

Existem duas maneiras de avaliar protocolos em ambiente assíncronos. Na dificuldade de determinar o número de *rounds* para a conclusão da instância de um protocolo, devido às características assíncronas do ambiente, uma das maneiras de comparar protocolos é pelo

número de passos por *round* (no nosso caso visão global). No nosso caso, como envolve o *Paxos* sem a parte da fase do *prepare*, são três passos: um com o *propose* do líder, outro com a troca dos *accepts* e o passo de envio dos *replies* (ignorando o passo do envio do cliente). Mas isto é no caso de operação normal sem ocorrência de falhas.

Certamente, o que nos preocupa é a ocupação de banda no suporte de rede. Ou seja, qual o custo do nosso protocolo em situações mais complexas. Normalmente, o custo assintótico de qualquer protocolo de acordo é de  $O(N^2)$  mensagens e o nosso não foge disto.

Mas, considerando que os nossos temporizadores vão aumentando em todos os decursos de prazo definidos no protocolo quando atravessam períodos assíncronos, vai chegar um momento em que estes prazos devem se estender por um ou mais períodos de sincronismo no sistema (períodos deterministas onde as instâncias de protocolo concluem). Neste caso, podemos assumir uma análise de pior caso na determinação da ocupação de banda do nosso protocolo.

A pior situação é quando numa instância temos troca de  $g$  sítios líderes e para que isto ocorra é necessário também  $f$  trocas de visões locais em cada sítio. Nesta situação estaríamos atingindo os limites de falhas admitidos no nosso sistema. Consideramos, para fins de análise, que o número de sítios no sistema é dado por  $N = 2g + 1$ .

O custo normal de ocupação de banda do nosso protocolo na execução de uma instância, sem trocas de visão, é dado por envolve o envio de  $2g + 1$  mensagens de *reqS<sub>k</sub>* (admissão da mensagem),  $2g + 1$  mensagens *propose<sub>k</sub>*,  $(2g + 1)^2$  mensagens de *accept<sub>k</sub>* e  $f + 1$  mensagens de *reply* ao cliente. Isto define um custo ( $C_n$ ) total de mensagens:

$$C_n = 2*(2g + 1) + (2g + 1)^2 + (f + 1)$$

Uma mudança de uma visão local provoca o envio de  $2g + 1$  mensagens de *newLocalView<sub>k</sub>* e as trocas de  $(2g+1)^2$  mensagens de *certMsg<sub>k</sub>*. Quando temos  $f$  trocas de visão local o custo de ocupação de banda por um sítio é de  $f*((2g + 1) + (2g + 1)^2)$  mensagens. Se considerarmos que, para se ter uma troca de visão global, precisamos que  $g + 1$  sítios experimentem este número de trocas de visão local, então o custo total sobe para  $(g + 1)*((2g + 1) + (2g + 1)^2)$  mensagens.

O custo para uma troca de visão global é de  $2g + 1$  mensagens *gviewChange<sub>k</sub>* e de  $(2g + 1)^2$  mensagens *certMsg<sub>k</sub>*. O custo total para  $g$  trocas de visão global é dado por:

$$g*(g + 1)*((2g + 1) + (2g + 1)^2)$$

Se ainda considerarmos que todos os sítios terminam as instâncias em configuração máxima e que a restauração exclui a coordenadora da configuração mínima. Então, teremos um custo adicional de  $2g + 1$

mensagens de  $newLocalView_k$  e de mais  $2g + 1$  mensagens de reconhecimento ( $ackniewLocal_k$ ) multiplicado pelo número de sítios no sistema ( $2g + 1$ ) na conclusão da instância. Com isto o custo total de trocas de visão em situação de pior caso é dado por:

$$C_T = g*(g + 1)*(f + 1)*((2g + 1) + (2g + 1)^2) + 2*(2g + 1)^2$$

Se somarmos este custo total com a execução normal teremos então o custo de conclusão em pior caso como dado por:

$$C_P = C_T + C_n$$

$$C_P = g*(g + 1)*(f + 1)*((2g + 1) + (2g + 1)^2) + 3*(2g + 1)^2 + 2*(2g + 1) + (f + 1)$$

Se consideramos  $g = 4$  e  $f = 1$  teremos um custo normal de:

$$C_n = 101 \text{ mensagens}$$

Para custo de pior caso:

$$C_P = 3863 \text{ mensagens}$$

Se usarmos para comparação o PBFT [Castro e Liskov, 1999], considerando para isto os seus passos em caso normal e em situação de  $g$  trocas de líder, são obtidos os seguintes custos:

$$C_n = 364 \text{ mensagens e}$$

$$C_P = 2236 \text{ mensagens.}$$

Em condições normais é obvio que a nossa proposta deveria ser mais eficiente do que o PBFT porque envolve menor número de trocas. Isto devido à existência de componente confiável na nossa abordagem. Mas, na condição de pior caso, a nossa proposta exige quase o dobro de mensagens das enviadas no PBFT na recuperação envolvendo  $g$  trocas de líder. A vantagem do nosso sistema é que este comporta o tratamento das réplicas maliciosas e que o número de intrusões é limitado por instâncias de protocolo, mas é ilimitada durante o ciclo de vida do servidor replicado.

## 6.2 PROTÓTIPO DESENVOLVIDO

Um protótipo foi desenvolvido para testar o comportamento do protocolo proposto. Os experimentos realizados visaram mostrar a viabilidade prática do protocolo proposto. Nesta seção apresentamos detalhes de nossa implementação e os resultados obtidos em testes de cenários que o protocolo foi exposto. Inicialmente, serão apresentados os detalhes do ambiente. As análises dos testes e seus resultados obtidos são apresentados na sequência desta seção.

### 6.2.1 Detalhes de Implementação do Protótipo

O ambiente de execução da implementação mostrado na Figura 6.6 ilustra as camadas que definem o nosso protótipo e a composição das máquinas físicas. Neste protótipo a ferramenta de virtualização (*Virtual Machine Monitor - VMM*) utilizada foi o *hypervisor* XEN [XEN, 2009], sendo executado sobre o sistema operacional Linux *OpenSuse* o qual dá suporte para a virtualização. Inicialmente a escolha desta ferramenta se deve pela infraestrutura desenvolvida para o protocolo *ITVM* descrito na seção 4.2. Além disso, esta ferramenta de virtualização nos permite a utilização de uma área de memória compartilhada que é oferecida pela própria ferramenta, chamada de *XenStore*. Este recurso oferece registradores de memória que são compartilhados entre as máquinas virtuais que as suprir as nossas necessidades de comunicação entre o *Agreement Service* (AS) e as máquinas virtuais (*VMs*) através de memória compartilhada. Além disso, o *VMM* também é responsável pelo controle de acesso a essa memória compartilhada. São definidas políticas de acesso restritivas de modo que cada réplica só pode ler e/ou escrever em registradores específicos, mantendo assim a consistência e a segurança da memória compartilhada.

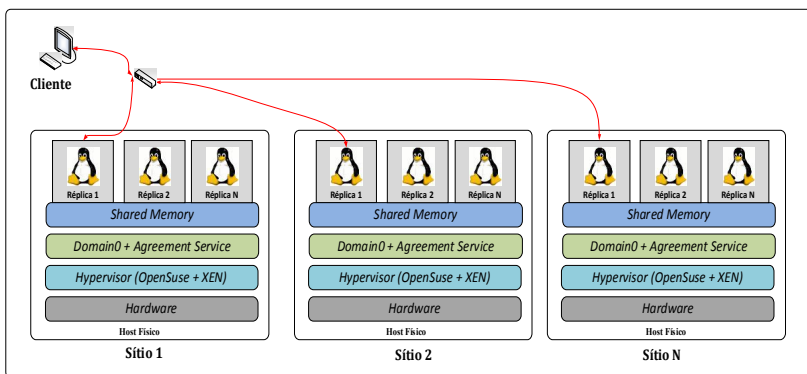


Figura 6.6 – Camadas do protótipo.

Assim como foi usado no protocolo *ITVM*, também fazemos uso de um domínio especial que é fornecido pelo *VMM*, chamado de *Domain0*. Este domínio único é utilizado para o gerenciamento das máquinas virtuais e da memória compartilhada. É através deste monitor de máquinas virtuais que podemos ativar e remover máquinas virtuais e concretizar o controle de acesso à memória compartilhada. É neste contexto que inserimos a nossa premissa do componente confiável no

*Domain0*. A característica própria da ferramenta de virtualização usada nos permite manter este domínio isolado, para atender a premissa assumida de componente confiável. Além disso, desabilitamos todos os acessos (portas) via rede de modo a manter o domínio inacessível via rede.

Com o objetivo de garantir os requisitos de diversidade de sistema operacional no protótipo os sistemas operacionais nas máquinas virtuais foram instalados diferentes tipos de sistemas operacionais Linux e com diferentes versões de *kernel*. Desta maneira, conseguimos uma maior independência de falhas ou intrusões de réplicas.

No protótipo usamos técnica de criptografia assimétrica nas mensagens trocadas entre o cliente e o *Agreement Service*. Utilizamos a criptografia RSA para a assinatura das mensagens trocadas entre as partes envolvidas. Nos testes realizados utilizamos uma chave de criptografia de tamanho 1024 bits.

### **6.2.2 Serviço de Aplicação com Replicação ME**

Em relação ao serviço de aplicação usamos a mesma definida para o teste do protocolo *ITVM* que é baseada em uma transação bancária no qual o cliente faz o envio das requisições com operação de crédito ou débito para uma réplica coordenadora do sítio do serviço replicado. A requisição do cliente é então encaminhada para um sítio que faz a admissão da requisição no sistema e encaminha para um líder que irá propor uma ordem de execução da mesma para os outros sítios. Esta proposta de ordem para a requisição uma vez aceita determina a execução da requisição em todos os sítios.

A execução da requisição faz com que o estado da aplicação seja modificado conforme o tipo de operação especificado na requisição. As réplicas, após executarem a operação, inserem a resposta na memória compartilhada. O *Agreement Service* aguarda por respostas iguais vindas de diferentes réplicas do sítio para criar a resposta a ser enviada para o cliente. Uma vez verificada esta condição, *replies* são enviados ao cliente no número de réplicas do sítio que recebeu a requisição do cliente.

Além disso, são executadas operações de *checkpoint* pelas réplicas com o objetivo de manter a consistência do estado da aplicação. Então, nestas operações, as réplicas salvam um *snapshot* das contas dos clientes. Em nossos testes, as operações de *checkpoint* foram efetuadas a cada cinquenta (50) requisições de cliente executadas pelas réplicas.

### 6.2.3 Testes sobre o Protótipo

No protótipo, cliente e servidores se comunicam através de uma rede local de 100 Mbps composta por uma máquina servidora (máquina física) com processador *Core I7*, 8 GB de RAM. Cada máquina virtual foi configurada neste servidor de forma a compartilhar o processador da máquina física, utilizando 1GB de memória RAM.

Nossos testes perderam em escala devido a dificuldades encontradas com ferramentas. Então os objetivos destes testes se reduziram a verificar o funcionamento do modelo de replicação tolerante a intrusão e a experimentar o comportamento com diferentes tipos de intrusões. Logo, em escala reduzida o nosso modelo de sistema o protótipo que é executado em dois níveis de protocolo de consenso, foi testado em limites reduzidos onde foi só possível verificar o comportamento do mesmo. No nível local, o limite assumido de réplicas maliciosas é de  $f = 1$ . Portanto, a configuração mínima local que inicia cada instância de protocolo, é de duas (2) réplicas. Este valor é razoável pelas condições de configuração dinâmica das réplicas de sítio. Tanto faz ter  $f = 1$  ou valores maiores deste  $f$  que as consequências são as mesmas. O desacordo de uma réplica na configuração mínima ( $f + 1$  réplicas) provoca a evolução para a configuração mínima. O que se perde é a capacidade de detecção de réplicas corrompidas atuando simultaneamente. Para o nível global, o número de sítios que podem apresentar falhas de *crash* foi limitado por  $g = 1$ . Como, neste nível, o comportamento faltoso é de *crashes*, na ocorrência de um desses *crashes*, o número de sítios para manter a redundância é 3 ( $2g + 1$ , com  $g = 1$ ). É normal e suportável. Estes valores de  $f$  e  $g$  foram escolhidos por limitações da máquina física hospedeira em manter máquinas virtuais.

Todos os testes foram executados em experimentos com a concorrência de 1000 requisições enviadas por vários clientes. Em cada teste específico, foram realizados três (3) experimentos de 1000 requisições. Com isto, consideramos sempre como resultado a média aritmética dos 3 experimentos.

O cálculo do tempo de respostas é feito no cliente, considerando o momento em que o mesmo envia a requisição até o momento em que recebe uma resposta válida vinda das réplicas do sítio (*Round-Trip Time*).

Os testes realizados foram executados em dois cenários distintos: (i) execução do protocolo sem a presença de réplicas maliciosas, e (ii) execução do protocolo na presença de réplicas maliciosas.



## 6.2.4 Experimentos com Execuções Normais de Protocolo

Neste cenário apresentamos a execução do protótipo em execuções normais, isto é, sem a ocorrência de faltas ou intrusões. O objetivo é mostrar o comportamento do protocolo e o tempo de resposta sendo executado no protótipo. Na Figura 6.7 ilustra o comportamento do sistema neste teste. No gráfico apresentado, utilizamos uma amostra de 500 requisições de um total 1000 amostras de dados obtidos dos testes.

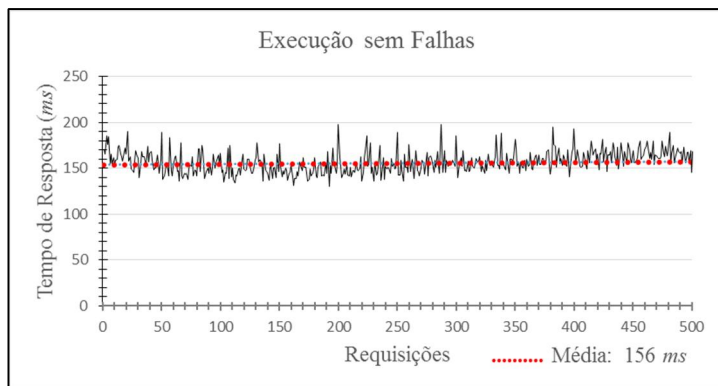


Figura 6.7 – Comportamento do sistema sem falhas.

Foi obtido também um histograma da distribuição geral dos tempos de respostas das requisições que é ilustrado na Figura 6.8. No eixo  $x$ , são apresentados tempos de resposta distribuídos em blocos de valores em milissegundos ( $ms$ ). O eixo  $y$  representa a frequência com que estes tempos de respostas aparecem dentro de um experimento de 1000 requisições. Podemos observar que há uma variação dos valores entre 135  $ms$  e 205  $ms$  o que corresponde a uma variação de 34,14%. No entanto, este valor está dentro do esperado, uma vez que, a grande maioria das requisições se encontra próxima da média que é de 156  $ms$ .

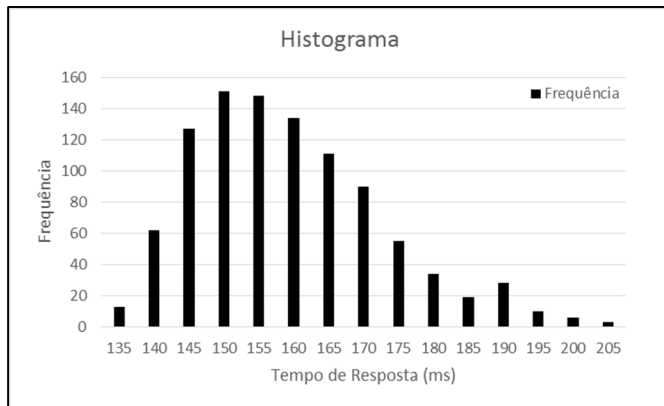


Figura 6.8 – Histograma dos testes sem falhas.

### 6.2.5 Experimentos com Simulação de Falhas no Protocolo

Nesta seção, mostramos execuções do protocolo com a presença de elementos corrompidos. O objetivo é mostrar o comportamento do protocolo quando se tem réplicas se comportando de maneira arbitrária, apresentando não somente respostas erradas, mas também se omitindo e atuando de maneira maliciosa, no intuito de impedir o correto funcionamento do protocolo.

Na literatura os trabalhos sobre faltas bizantinas apresentam o limite máximo de réplicas maliciosas fixo (em  $f$  falhas). Isto evidentemente permite que o protocolo se mantenha em progresso de sua execução. No entanto, as falhas que vão ocorrendo no ciclo de vida destas propostas, reduzem a capacidade de tolerar intrusões e falhas. Não existe recuperação automática dos elementos corrompidos (faltosos) e com isto robustez do sistema vai sendo reduzida. Além disso, quando esse limite de  $f$  falhas é alcançado durante a sua execução e qualquer outra falha subsequente no sistema faz com que o serviço deixe de funcionar corretamente. Desta maneira, é necessário que se tenha um mecanismo para o tratamento de réplicas maliciosas ou o rejuvenescimento das réplicas que atue dinamicamente no sistema e constantemente.

Em nossa proposta as réplicas maliciosas são removidas, de modo que, o número de réplicas corretas sempre inicie uma execução de protocolo em configuração mínima ( $f + 1$  réplicas). E evolui em situação de desacordo para a configuração máxima ( $2f + 1$ ). A ME é sempre restaurada na sua configuração mínima de  $f + 1$  réplicas no término do acordo. Em relação aos crashes em nível global, a troca de visão global

(troca de líder) que é a única ação que realizamos neste nível, é efetuada a cada  $f$  trocas de visão local em  $g + 1$  sítios. Neste nível não ocorre o tratamento dinâmico de elementos faltosos.

Para verificar o comportamento do protocolo, foram montados alguns casos de testes com diferentes distribuições de réplicas maliciosas. Na Tabela 6.1 são apresentadas essas distribuições de faltas. Na primeira coluna ( $f\%$ ), temos o valor percentual de falhas que ocorrem dentro de um conjunto de 1000 requisições. A segunda coluna ( $N_f$ ) representa o número total de falhas que ocorrerão durante as 1000 execuções. E por fim, a coluna ( $\Delta r$ ) representa o intervalo de requisições corretas até que uma execução faltosa ocorra. Nos experimentos optamos em simular as faltas em requisições pré-definidas para facilitar a visualização dos resultados.

Tabela 6.1 – Distribuição das falhas

$f\%$	$N_f$	$\Delta r$
0,1 %	1	*
1 %	10	99
5 %	50	19
10 %	100	9
25 %	250	3
50 %	500	1

Nas subseções subsequentes, são apresentados quatro experimentos com comportamentos maliciosos que podem ocorrer durante a execução do protocolo. O primeiro comportamento malicioso é o caso onde o sítio que recebeu a requisição do cliente não envia a mensagem *reqS* para os outros sítios. No segundo comportamento simulado, o sítio líder não envia a mensagem *propose* para os outros sítios. O terceiro comportamento é o caso quando um sítio não recebe um número mínimo de mensagem *accept* de diferentes sítios e por último é o caso quando se tem um desacordo das respostas.

## 6.2.6 Comportamento com Falha no Envio de Mensagem de Admissão

### Experimento com poucas Falhas (0,1%)

No cenário com apenas uma falha (Figura 6.9 e Figura 6.10) o coordenador do sítio receptor de uma requisição do cliente não envia a

mensagem *reqS* aos demais sítios. Estas falhas estão distribuídas nas requisições, ou seja, uma em cada mil requisições vão apresentar problemas na sua evolução. E as requisições geradas no sistema são endereçadas a qualquer sítio da ME. As execuções de requisições apresentam em problema em 0,1 % destas execuções.

É possível observar que devido ao baixo número de falhas e mesmo com o acréscimo de tempo, devido ao intervalo entre a ocorrência da falha da réplica receptora (da coordenadora do sítio receptor da requisição) e mais a troca de visão local de seu sítio para a recuperação, o tempo de resposta médio no conjunto de requisições do experimento sofre pequena mudança de valor (157 ms). O impacto gerado por esta falha não chega a interferir no tempo de resposta do conjunto de requisições executados no experimento como um todo. Mas, atrasa as réplicas que entregam os *replies* ao cliente no valor indicado. Na Figura 6.10 representa a distribuição dos tempos de respostas executados neste cenário.

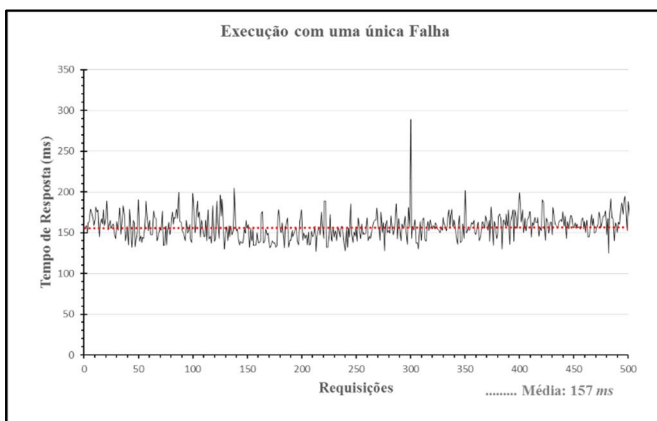


Figura 6.9 – Gráficos da execução com 0,1 % de instâncias com falhas.

Partindo do pressuposto que os dados obtidos possuem uma distribuição normal, pode-se determinar intervalos de tolerância para os tempos de respostas. Se considerarmos o intervalo de confiança correspondente a 68,27% podemos garantir que as distribuições dos tempos de respostas podem ser obtidas entre 141 ms e 173 ms, como mostrado na Tabela 6.2. Se aumentamos o intervalo de confiança para 86,64% podemos garantir que o tempo de resposta estará entre 134 ms a 181 ms.



Figura 6.10 – Histograma com 0,1 % de instâncias com falhas

Tabela 6.2 - Intervalo de confiança para a execução com 0,1% de falhas

Com 0,1% de Falhas		
<b>Média (<math>\mu</math>)</b>	157	
<b>Desv. Pad. (<math>\sigma</math>)</b>	16	
<b>Intervalos de Tolerância de Desvio</b>	<b><math>\mu \pm \sigma</math> (68,27%)</b>	<b><math>\mu \pm 1.5\sigma</math> (86,64%)</b>
<b>Mínimo</b>	141	134
<b>Máximo</b>	173	181

#### Execução de experimento com 1% das Falhas

Neste experimento, apresentado na Figura 6.11 e Figura 6.12, representa um comportamento onde a ocorrência de requisições em que ocorrem falhas se torna mais constantes através picos que se mostram com mais frequência. Tempo médio de resposta envolvendo todas as requisições do experimento apresenta um pequeno aumento de 3 ms devido a essas falhas.

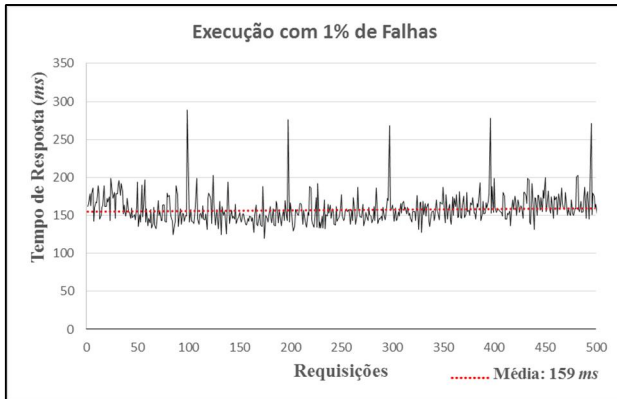


Figura 6.11 – Execução com 1% de falhas no envio da mensagem reqS

Aplicando a mesma análise realizada no experimento anterior, podemos observar na Tabela 6.3 que se considerarmos o intervalo de confiança de 68,27%, a distribuição dos tempos de respostas pode variar entre 138 ms e 180ms. Enquanto, para o intervalo de confiança os tempos de respostas fica entre 128 ms e 190 ms.

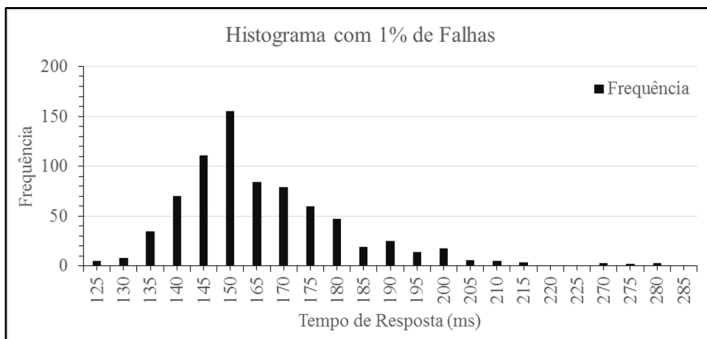


Figura 6.12 – Histograma dos testes com 1% falha

Tabela 6.3 – Intervalo de confiança para a execução com 1% de falhas

Com 10% de Falhas		
<b>Média (<math>\mu</math>)</b>	159	
<b>Dev. Pad. (<math>\sigma</math>)</b>	21	
<b>Intervalos de Tolerância de Desvio</b>	<b><math>\mu \pm \sigma</math> (68,27%)</b>	<b><math>\mu \pm 1.5\sigma</math> (86,64%)</b>
<b>Mínimo</b>	138	128
<b>Máximo</b>	180	190

### Experimento com 50% Falhas

Neste experimento, a presença de requisições onde instâncias do protocolo devem apresentar falhas é bem numerosa, como mostrado na Figura 6.13 onde uma em cada duas requisições está associada à ocorrência de falhas em suas instâncias de protocolo. Devido a estas altas frequências, o tempo médio de resposta envolvendo as mil requisições do experimento apresenta um aumento considerável, chegando a 231 *ms*. A distribuição dos tempos de respostas visto no histograma (Figura 6.14) difere dos histogramas anteriores, mostrando uma distribuição mais visível de execuções com falhas e sem falhas.

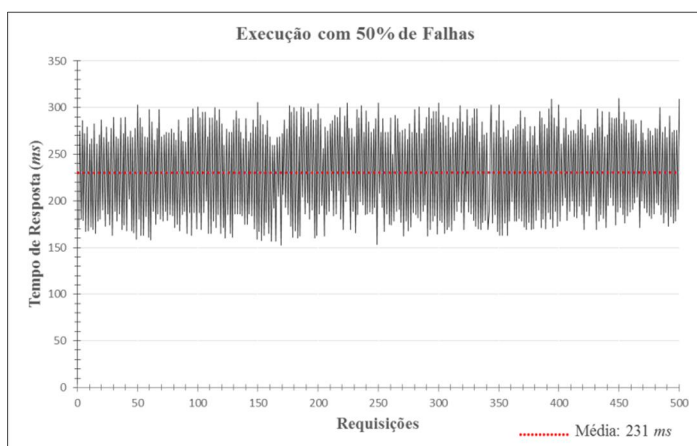


Figura 6.13 – Execução com 50% de falhas

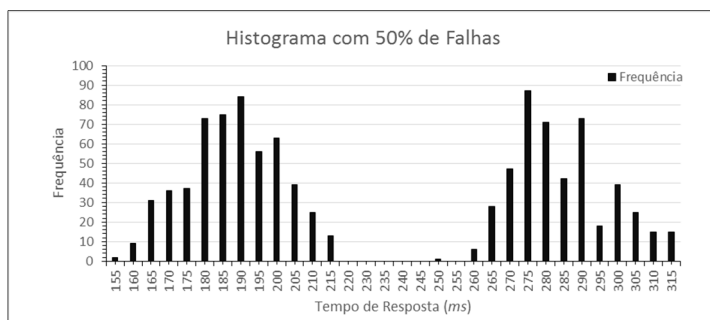


Figura 6.14 – Histograma dos testes com 50% de falhas

Aplicando a mesma análise realizada para a escolha de intervalos de tolerância, podemos observar na Tabela 6.4 que, se escolhido o intervalo de confiança de 68,27%, os tempos de respostas podem ser

obtidos entre 181 *ms* e 281 *ms*. Se escolhido o intervalo de confiança de 86,64 %, podemos garantir que esses valores estarão entre 156 *ms* e 306 *ms*.

Tabela 6.4 – Intervalo de confiança para a execução com 50% de falhas

Com 50% de Falhas		
Média ( $\mu$ )	231	
Desv. Pad. ( $\sigma$ )	50	
Intervalos de Tolerância de Desvio Padrão	$\mu \pm \sigma$ (68,27%)	$\mu \pm 1.5\sigma$ (86,64%)
Mínimo	181	156
Máximo	281	306

### 6.2.7 Comportamento com Falha no Envio de Mensagem *Propose*

#### Experimento com poucas falhas (0,1%)

Este cenário representa o pior cenário de execução com falha no protocolo. Este corresponde à situação em que a coordenadora do sítio líder, ao receber uma mensagem *reqS* não entrega a mesma ao seu *Agreement Service* e este não elabora a mensagem *propose* correspondente. Com isto, a coordenador não envia o *propose* para os demais sítios. Os sítios que tiverem recebido a mensagem de admissão da requisição do cliente, ficam aguardando o recebimento da mensagem *propose* com prazo definido. Após a ultrapassagem deste prazo (*prazoCoord*) sem o recebimento desta mensagem, trocam os seus coordenadores (trocas de visão local). Se ainda assim este problema persistir e ninguém receber a mensagem *propose* a suspeita recai no sítio líder que pode estar em *crash* e então é provocada uma troca de visão global. Desta forma, este experimento visa observar o comportamento dessas trocas dos coordenadores de cada sítio e da troca de líder global.



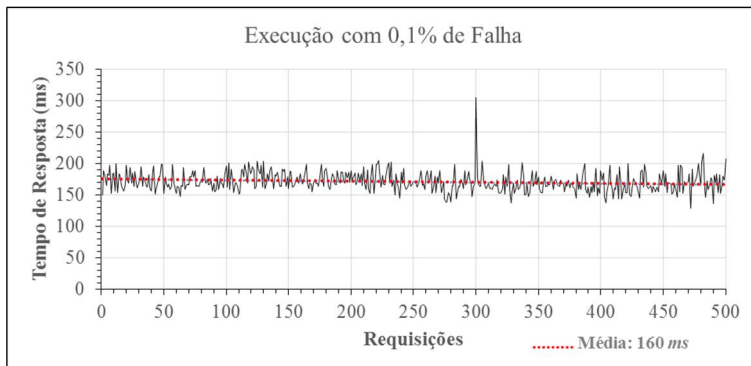


Figura 6.15 – Execução com 0,1% de Falha.

Para a execução com apenas 0,1 % de requisições apresentando este problema de comportamento, podemos observar na Figura 6.15 que o tempo médio de respostas não muda muito, ficando em 160 ms. Na Figura 6.16, é representado a distribuição dos tempos de respostas obtidos neste cenário.



Figura 6.16 – Histograma dos testes com 0,1% de falha.

Com relação ao intervalo de tolerância podemos observar na Tabela 6.5 que se considerarmos o intervalo de confiança de 68,27% a faixa de distribuição dos tempos de respostas fica entre 144ms e 177ms e para o intervalo de 86,64% ficar entre 136 ms e 185 ms. Os resultados obtidos nos mostram que a baixa frequência neste cenário não influencia muito nos resultados.

Tabela 6.5 – Intervalo de confiança para a execução com 0,1% de falha.

Com 0,1% de Falhas		
<b>Média (<math>\mu</math>)</b>	160	
<b>Dev. Pad. (<math>\sigma</math>)</b>	16	
<b>Intervalos de Tolerância de Desvio Padrão</b>	<b><math>\mu \pm \sigma</math> (68,27%)</b>	<b><math>\mu \pm 1.5\sigma</math> (86,64%)</b>
<b>Mínimo</b>	144	136
<b>Máximo</b>	177	185

### Experimento com 1% de Falhas

Neste cenário, apresentado na Figura 6.17, os picos começam a se torna mais constantes e o tempo médio de resposta, nas execuções das mil requisições do experimento com esta distribuição de 1%, a começa apresentar aumento considerável em relação a execução, apresentado como valor 176 ms. Os picos de tempo de resposta mostrados na Figura 6.18 ficam entre 305 ms e 335 ms e estes correspondem aos tempos de respostas nas instâncias de protocolo que sofrem as duas trocas de visões (uma local e outra global) previstas neste teste, no começo desta seção. Na

Tabela 6.6 são mostrados o intervalo de tolerância do desvio padrão para o tempo de resposta, considerando o intervalo de confiança entre 68,27% e 86,64%.

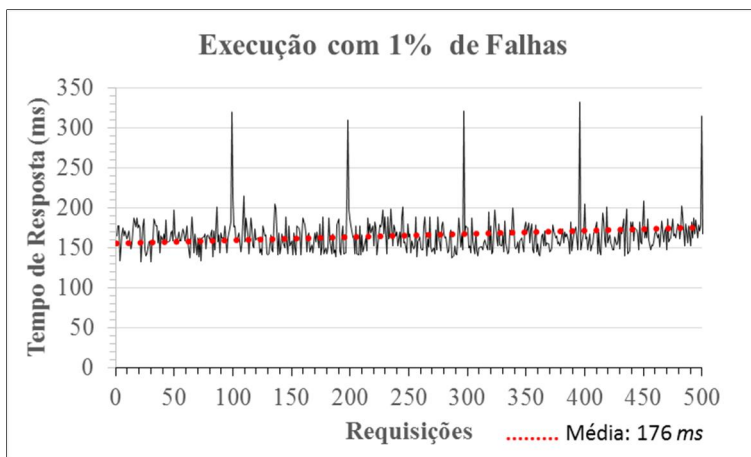


Figura 6.17 – Execução com 1% de falhas.

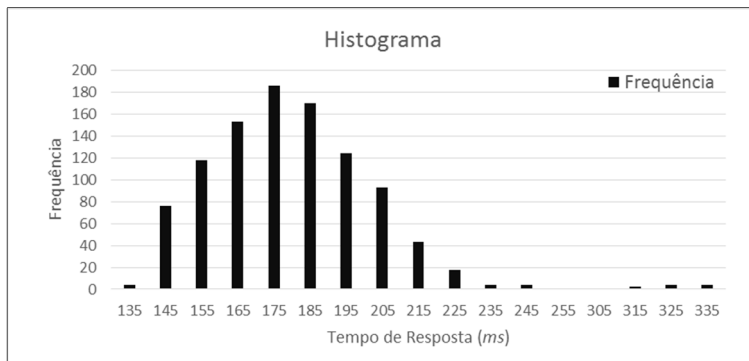


Figura 6.18 – Histograma dos testes com 1% de falhas.

Tabela 6.6 – Intervalo de confiança para a execução com 1% de falhas.

Com 1% de Falhas		
<b>Média (<math>\mu</math>)</b>	176	
<b>Desv. Pad. (<math>\sigma</math>)</b>	26	
<b>Intervalos de Tolerância de Desvio Padrão</b>	<b><math>\mu \pm \sigma</math> (68,27%)</b>	<b><math>\mu \pm 1.5\sigma</math> (86,64%)</b>
<b>Mínimo</b>	150	136
<b>Máximo</b>	202	215

### Experimento com 50% de Falhas

Neste experimento, podemos observar na Figura 6.19 que o tempo médio de resposta em relação às requisições do experimento, considerando 50% destas sofrendo falhas em suas instâncias de protocolo, sofreu um aumento de 38,69%, se considerado o tempo médio do teste anterior (teste com 1%). Isto acontece porque uma entre duas requisições terão nas suas instâncias de protocolo trocas de coordenador e de sítio líder. A Figura 6.19 mostra o histograma da distribuição dos tempos de respostas para essa execução. O gráfico mostra o custo dessas trocas de visão explicitado em tempos de respostas variando entre 275 ms a 345 ms. Ou seja, os tempos de respostas já manifestam as duas trocas de visão para a ausência de mensagem *propose*.

Na Tabela 6.7, nota-se que o intervalo de confiança para 68,27% não abrange os tempos de respostas citados uma vez que, devido ao aumento no número de requisições sob falhas de suas instâncias, os tempos de respostas ficam mais distribuídos. No entanto, se aumentarmos esse intervalo para 86,64% podemos notar que os tempos de respostas ficam entre 181 *ms* e 341 *ms*.



Figura 6.19 – Execução com 50% de falhas.

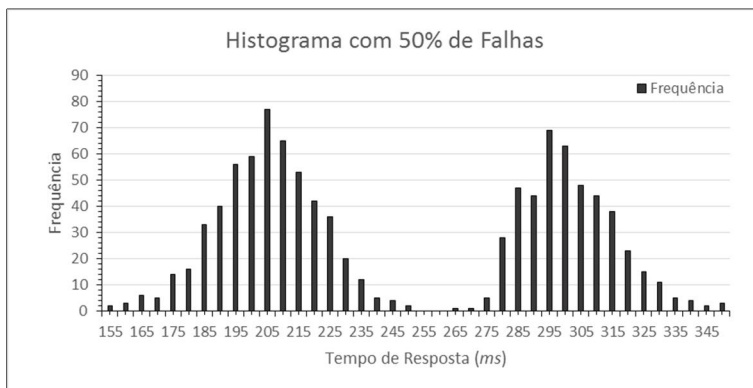


Figura 6.20 – Histograma dos testes com 50% de falhas.

Tabela 6.7 – Intervalo de confiança para a execução com 50% de falhas.

Com 50 % de Falhas		
<b>Média (<math>\mu</math>)</b>	261	
<b>Desv. Pad. (<math>\sigma</math>)</b>	53	
<b>Intervalos de Tolerância de Desvio Padrão</b>	<b><math>\mu \pm \sigma</math> (68,27%)</b>	<b><math>\mu \pm 1.5\sigma</math> (86,64%)</b>
<b>Mínimo</b>	208	181
<b>Máximo</b>	314	341

## 6.2.8 Comportamento em Falhas no Envio da Mensagem *Accept*

### Experimento com poucas falhas (0,1%)

Para a execução deste experimento foi considerado 0,1% das requisições do experimento tendo suas instâncias (de protocolo) aumentadas, devido ao fato que uma réplica (sítio) não conseguiu o número mínimo de mensagens *accept's* para dar o prosseguimento de sua execução. Desta maneira, novamente é feita uma troca do coordenador local e trocas de mensagens de sincronização que devem recuperar o sítio desatualizado. Nas Figura 6.6 e Figura 6.22 são mostrados o comportamento do protocolo com relação aos tempos de respostas quando é feito a troca de um coordenador de um sítio e na Tabela 6.8 são apresentados os intervalos de tolerância do desvio padrão para a chegada dos tempos de respostas.

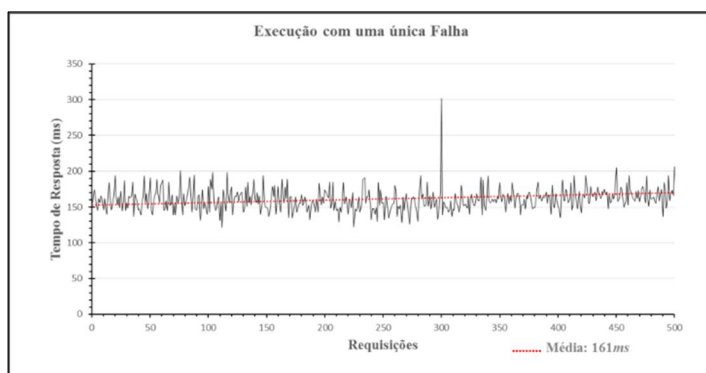


Figura 6.21 – Execução com 0,1 % de instâncias com falhas.

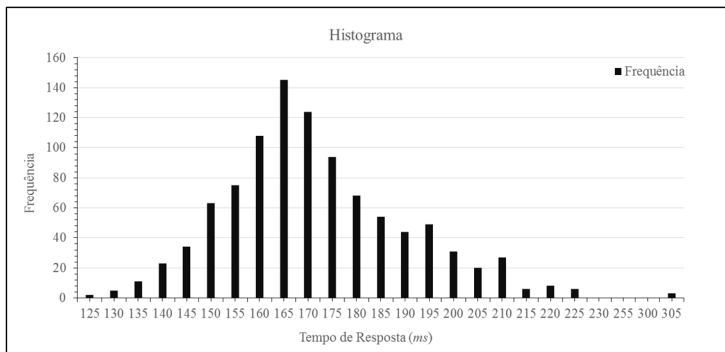


Figura 6.22 – Histograma dos testes com 0,1 % de instâncias com falhas.

Tabela 6.8 – Intervalo de confiança para a execução com 0,1% de falhas.

Com 0,1% de Falhas		
<b>Média (<math>\mu</math>)</b>	161	
<b>Desv. Pad. (<math>\sigma</math>)</b>	20	
<b>Intervalos de Tolerância de Desvio</b>	<b><math>\mu \pm \sigma</math> (68,27%)</b>	<b><math>\mu \pm 1.5\sigma</math> (86,64%)</b>
<b>Mínimo</b>	141	130
<b>Máximo</b>	181	191

### Experimento com 1% de Falhas

Para este experimento, já existe um aumento de requisições com instâncias de protocolo tratando exceções. Picos já começam a aparecer com maior frequência e o tempo médio de resposta das instâncias de protocolo começa a ser afetado devido à ocorrência de falha indicada para o teste, como mostrado na Figura 6.23 e Figura 6.24. Esse aumento, ainda que significativo, foi de 6,93% com relação ao experimento anterior. Na Tabela 6.10, é mostrado o intervalo de tolerância do valor máximo e mínimo para a entrega das respostas considerando o intervalo de confiança entre 68,22% e 86,64%.



Figura 6.23 – Execução com 1% de falhas.

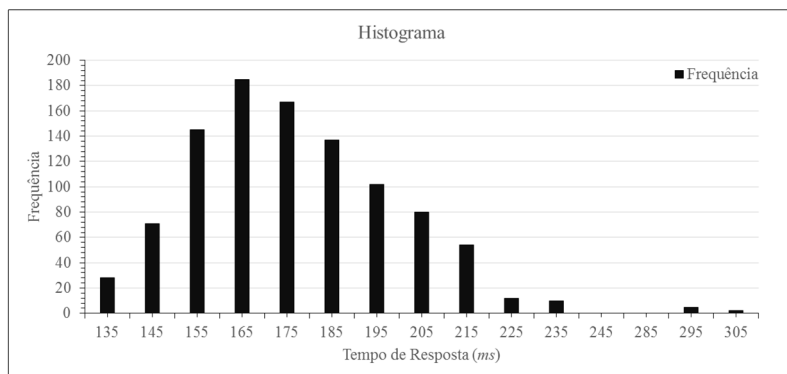


Figura 6.24 – Histograma dos testes com 1% de falhas.

Tabela 6.9 – Intervalo de confiança para a execução com 1% de falhas.

Com 1% de Falhas		
Média ( $\mu$ )	173	
Dev. Pad. ( $\sigma$ )	25	
Intervalos de Tolerância de Desvio	$\mu \pm \sigma$ (68,27%)	$\mu \pm 1.5\sigma$ (86,64%)
	Mínimo	Máximo
	148	136
	198	211

## Experimento com 50% de Falhas

Este cenário, com uma ocorrência bem maior de requisições cujas instâncias experimentam falhas com as mensagens *accept*, como evidenciado na Figura 6.25, o tempo médio de resposta é de *257 ms*, um aumento de 39,29% em relação a uma execução sem falha. A anomalia que as instâncias experimentam é só de uma troca de coordenador local e apenas em um sítio. Este caso é menos penalizante que o do teste da mensagem *propose* com 50% de requisições com problemas. Aquele caso anterior envolvia duas trocas de visão em todos os sítios. Em relação ao intervalo de tolerância, mostrado na Tabela 6.10, podemos garantir a entrega das mensagens considerando os intervalos de confiança para 68,22% e 86,64%.

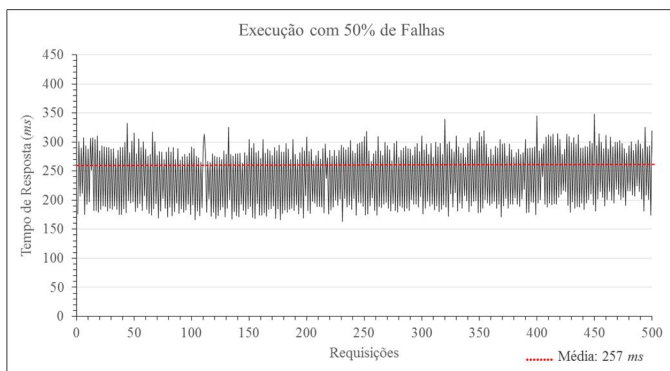


Figura 6.25 – Execução com 50% de falhas.

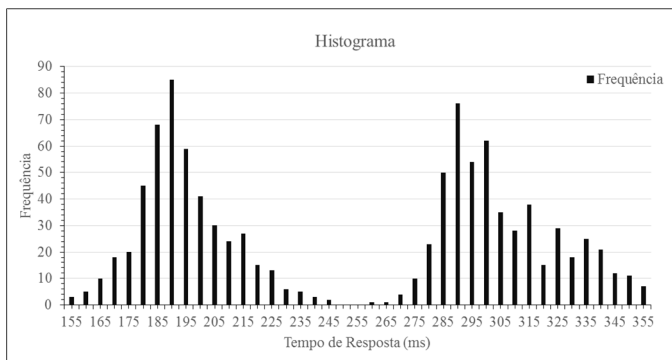


Figura 6.26 – Histograma dos testes com 50% de falhas.



Tabela 6.10 – Intervalo de confiança para a execução com 50% de falhas.

Com 50% de Falhas		
Média ( $\mu$ )	257	
Desv. Pad. ( $\sigma$ )	56	
Intervalos de Tolerância de Desvio Padrão	$\mu \pm \sigma$ (68,27%)	$\mu \pm 1.5\sigma$ (86,64%)
Mínimo	201	173
Máximo	313	341

## 6.2.9 Comportamento em Desacordo de Respostas de Réplicas Locais

### Experimento com poucas falhas (0,1%)

Neste experimento, é representado o caso em que uma réplica maliciosa de um sítio tenta causar um desacordo com as respostas das réplicas do mesmo sítio. Neste caso, o *Agreement Service* ao detectar um desacordo ele ativa configuração máxima para solucionar o conflito das respostas. Este experimento tenta avaliar o custo deste comportamento errôneo no sistema. Na Figura 6.27 é possível perceber o momento desta falha maliciosa uma intrusão até a recuperação do sítio e que também o tempo de resposta neste caso sofre um aumento considerável 1s (1000 ms). Este valor se deve principalmente pela inicialização e recuperação do estado pela réplica que está sendo adicionada ao protocolo. Mas, neste caso, o impacto não chega a interferir no conjunto como um todo.



Figura 6.27 – Execução com 0,1 % de instâncias com falhas.

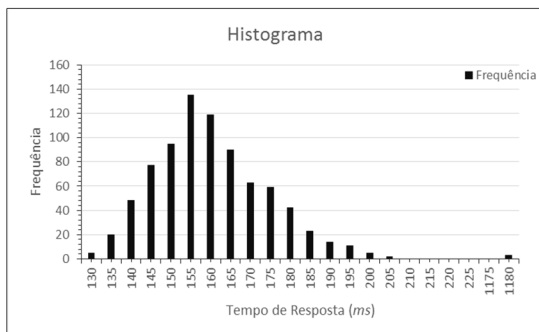


Figura 6.28 – Histograma dos testes com 0,1 % de instâncias com falhas.

### Experimento com 1% de Falhas

Para este caso, com 1% de instâncias de requisições tratando com falhas, os tempos de respostas da ME começam a ser afetados pela presença já observável de instâncias com falhas (Figura 6.29 e Figura 6.30). Os picos mostram os custos das inicializações e recuperações após a falha provocada pela réplica corrompida no protocolo. Podemos perceber que já se tem um aumento visível do tempo médio de resposta se comparado com o teste anterior de 0,1%.

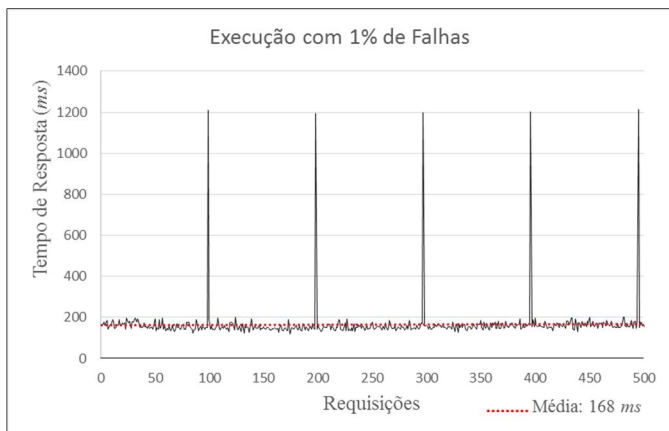


Figura 6.29 – Execução com 1 % de falhas.

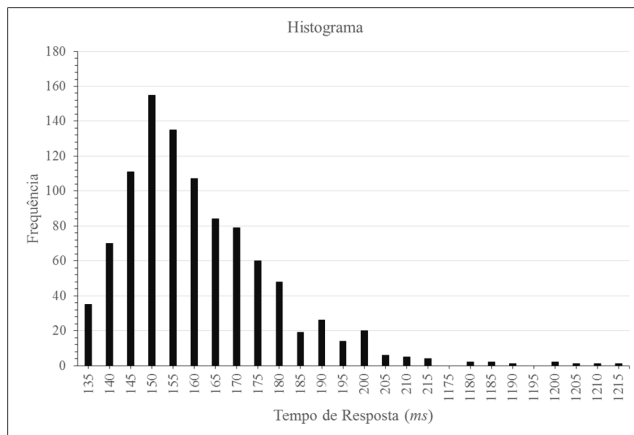


Figura 6.30 – Histograma dos testes com 1% de falha.

### Experimento com 50% de Falhas

Neste experimento, apresentado na Figura 6.31, a presença de falhas traz um valor bastante elevado devido a frequência alta da ocorrência de instâncias com problemas de réplicas maliciosas. Tendo apenas uma execução correta para cada execução com falha. O tempo médio de resposta em um experimento de 1000 requisições é quatro vezes maior se compararmos como o cenário de execuções normais. É possível observar na Figura 6.32 que a distribuição entre as instâncias que tratam estas falhas maliciosas e as que evoluem sem problemas.

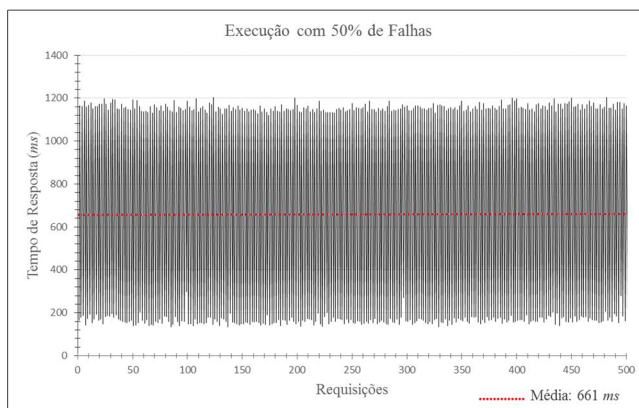


Figura 6.31 – Execução com 50 % de falhas.

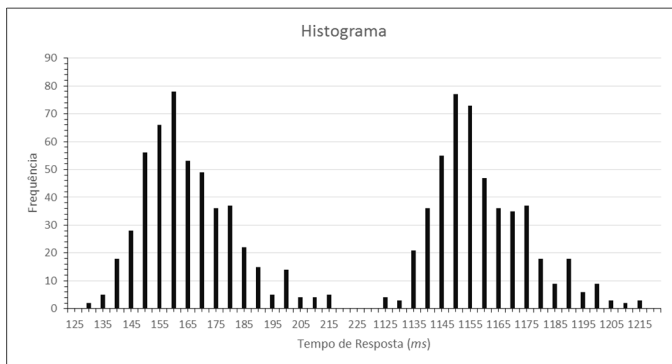


Figura 6.32 – Histograma dos testes com 50% de falhas.

### 6.2.10 Impacto das Falhas no Tempo Médio de Resposta

Durante as seções anteriores, foram apresentados os gráficos de execução e histogramas dos tempos de respostas de diferentes cenários. Ainda assim, faltaram acrescentar alguns cenários com distribuição de instâncias com anomalias citados na seção 6.2.5. Não apresentamos os mesmos (5%, 10% e 25% de falhas) para evitar textos repetitivos. Mas, a Figura 6.33 mostra estes testes junto com os outros já descritos neste capítulo.

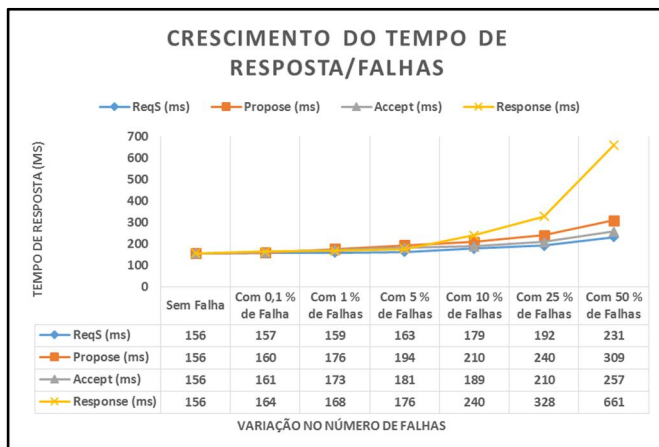


Figura 6.33 – Tempo de resposta o aumento progressivo de falhas.

Esta figura evidencia o crescimento do número de falhas em experimentos, partindo do caso de execuções normais e indo até o caso com falhas em 50% das requisições. Considerando então muitos dos cenários já apresentado e estes só citados, verificamos que o tempo médio de resposta tem um aumento variando entre 32% e 40% com o aumento das taxas de anomalias. A exceção para esta constatação é o caso em que se tem um desacordo de réplica maliciosa. As razões para isto já foram comentadas e estão ligados aos custos de ativação das novas réplicas.

### 6.2.11 Throughput

Para avaliar o *throughput* de nosso protótipo, a Figura 6.34 apresenta um gráfico resumido *throughput* com diferentes cenários de testes propostos. Assim como nos resultados apresentados anteriormente, é possível perceber que o aumento do número de intrusões tem grande impacto no tempo de resposta do sistema, e por consequência, em seu *throughput*. Nota-se que a medida que o número de falhas vai aumentando o *throughput* também diminui.

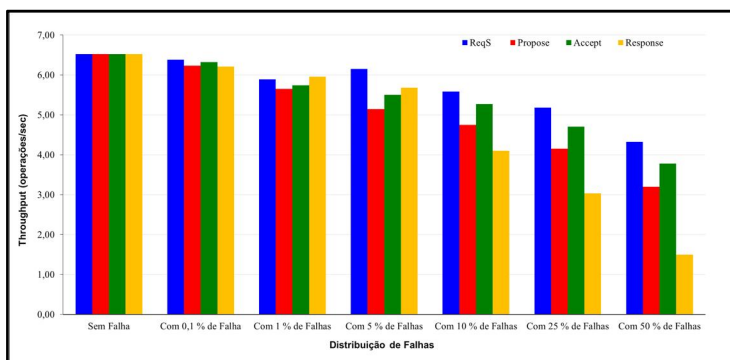


Figura 6.34 – *Throughput* em operação sem falha e com falhas.

### 6.2.12 Clientes Simultâneos

Para avaliar o desempenho do protocolo sob situação de alta carga, foi feita uma simulação utilizando vários clientes enviando requisições simultaneamente. Desta forma, na Figura 6.35 pode-se perceber um impacto no tempo de resposta, pois, com o aumento do número de clientes, a utilização e a concorrência sobre os recursos da máquina física são incrementados. Porém, mesmo em sistemas onde não há qualquer mecanismo para o tratamento de faltas, este efeito de aumento no tempo de resposta a requisição também ocorre.

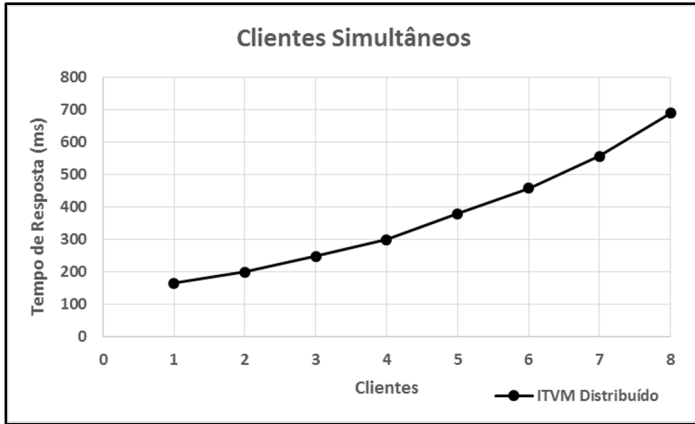


Figura 6.35 – Clientes Simultâneos.

### 6.2.13 Distribuição da Latência

Para avaliar o comportamento do protocolo ITVM Distribuído em uma WAN foi feita uma simulação usando diferentes valores de latência. Esses valores são baseados a partir de testes feitos em uma rede real. Isto é, por exemplo, na Tabela 15 mostra diferentes

Tabela 6.11– Latência da rede.

Locais	Latência (ms)
Florianópolis	2
Porto Alegre	7
Minas Gerais	23
Argentina	46
Chile	62
Itália	263
Portugal	259
Espanha	266
Suécia	322
EUA	156
Canadá	268

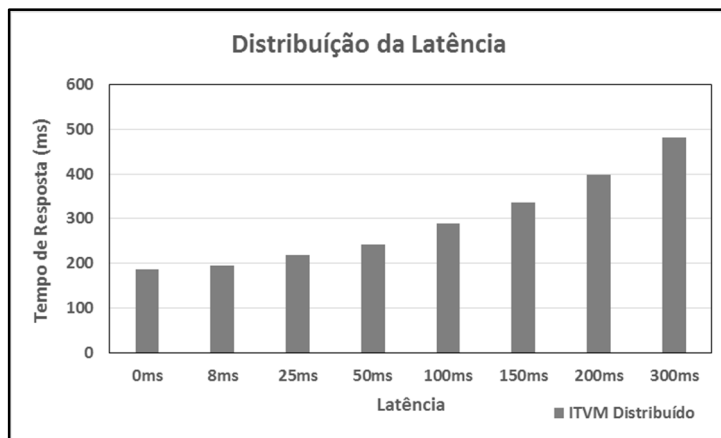


Figura 6.36 – Distribuição da Latência

### 6.2.14 Considerações sobre o Protótipo

Durante o desenvolvimento do protótipo foram feitas observações importantes. A utilização da linguagem de programação Java 7 na implementação do nosso protótipo, que nos permitiu a criação deste protótipo de forma mais ágil, em virtude de uma grande quantidade de pacotes e bibliotecas prontas. No entanto, o fato de usar essa linguagem de programação integrada junto com a memória compartilhada oferecida pelo *hypervisor* XEN causou um *overhead* considerável. Uma vez que, essa tecnologia é implementada em outra linguagem (linguagem C e *Python*). Os mecanismos de acesso aos recursos de memória compartilhada são somente implementados com as últimas linguagens citadas. Com isso, grande parte do impacto nos tempos de resposta obtidos deve ser creditado às operações de leitura e escrita em memória compartilhada.

Acreditamos que o uso de outros modelos de memória compartilhadas ou a implementação do protótipo com a mesma

linguagem usada pelo XEN poderia possibilitar uma melhora significativa no desempenho do protótipo apresentado.

Outro detalhe observado foi que nos testes realizados a execução da virtualização foi feita em máquinas comuns. A utilização de hardwares específicos para a virtualização também pode gerar um ganho considerável no desempenho do nosso protótipo.

### 6.3 COMPARAÇÕES DE NOSSAS PROPOSIÇÕES COM TRABALHOS RELACIONADOS

Na literatura são encontrados alguns trabalhos utilizando o conceito de virtualização como suporte para algoritmos distribuídos tolerantes a faltas bizantina. Entre estes trabalhos temos [Reiser e Kapitza, 2007] [Júnior, *et al.*, 2010] [Wood, *et al.*, 2011]. Em [Reiser e Kapitza, 2007] é proposta a execução de um serviço replicado implementado sobre várias máquinas virtuais também em um único sistema físico. Além disso, esta proposta faz uso de um componente confiável para auxiliar na execução do algoritmo. Entretanto, este componente se encontra exposto na rede possibilitando uma invasão, o que pode comprometer o correto funcionamento do modelo.

Em [Júnior, *et al.*, 2010] é proposto a arquitetura SMIT e um algoritmo onde as réplicas de serviço são executadas em máquinas virtuais e a comunicação entre as réplicas é realizada através de uma abstração de uma memória compartilhada. Este modelo, pelo uso de memória compartilhada e elemento confiável tem também a redução no número necessário de réplicas de  $3f + 1$  para  $2f + 1$  quando em comparação com PBFT executados sobre redes locais [Castro e Liskov, 1999]. Em nosso modelo utilizamos uma abordagem semelhante, porém reduzimos o número de réplicas para  $f + 1$  em situações favoráveis (sem malícia) devido ao uso do *Agreement Service* no processo de votação e da assinatura do mesmo na resposta acordada. Quando ocorrem réplicas maliciosas no nosso modelo o *Agreement Service* (AS) detecta as mesmas pelo desacordo nas respostas. Ativa mais  $f$  réplicas e lê as respostas destas últimas para completar  $f + 1$  respostas concordantes. Na sequência, o AS remove as réplicas discordantes e os excedentes retornando o sistema para a configuração de  $f + 1$  réplicas. Depois disto, o AS libera as respostas concordantes devidamente assinadas para que as réplicas que ficaram na configuração enviem estas ao cliente. O nosso modelo, portanto, regenera a replicação em cada execução de requisição. No SMIT, as intrusões vão sendo acumuladas no ciclo de vida do sistema. Não existe recuperação da replicação diante de réplicas maliciosas.



Em [Wood, *et al.*, 2011] é proposto a arquitetura ZZ e um protocolo utilizando a tecnologia de virtualização para execução de um protocolo BFT no qual aplica o conceito de separação do protocolo de acordo e da execução definida por [Yin, *et al.*, 2003]. Neste trabalho existe também a alteração dinâmica da replicação do modelo, com a replicação também assumindo configurações de  $f + 1$  e  $2f + 1$  réplicas. As diferenças do ZZ com a nossa abordagem de BFT é que as VMs de réplicas no ZZ são executadas em um conjunto de máquinas físicas. O ZZ não trabalha com a ideia de elementos confiáveis como o AS do nosso modelo e, portanto, não permite a separação de faltas de *crash* e de intrusões. No ZZ, as VMs não interagem via memória compartilhada, mas sim usando troca de mensagens via protocolos de rede o que implica em custo adicional de interações entre VMs.

Em relação aos trabalhos envolvendo serviços *web* e replicação Máquina de Estados para a tolerância a intrusões, temos as propostas de [Zhao, 2008] e [Merideth, *et al.*, 2005]. Estes trabalhos introduzem infraestruturas visando à tolerância a intrusões em provedores de *serviços web*. Estes trabalhos são baseados no PBFT [Castro e Liskov, 1999] onde utilizam  $3f+1$  réplicas para garantir a execução do algoritmo de acordo. Nos trabalhos citados, cada réplica de serviço é executada em uma máquina física no qual eleva o custo de mensagens da replicação para  $O(N^2)$ , onde o  $N$  é o número de réplicas nos modelos. No nosso modelo, o custo assintótico do nosso modelo é o mesmo devido ao *Paxos* usado entre sítios. Mas o custo para tolerar intrusões é bem mais simples devido às comunicações via memória compartilhada.

Na dificuldade de evitar acessos ilegais, muitas abordagens têm sido apresentadas na literatura no sentido de conter ou restringir as possíveis intrusões. Neste texto, centramos nosso interesse nos modelos de tolerância a intrusões para sistemas distribuídos [Correia, 2005]. A ideia, neste caso, é manter o sistema distribuído evoluindo com comportamento correto mesmo diante de alguns componentes já corrompidos por intrusões. A ação maliciosa destes componentes invadidos não deve afetar o sistema como um todo. Nossos desenvolvimentos tiveram sempre como objetivo seguir esta linha, mas tentando reduzir custos.

Tabela 6.12 - Comparativo entre as diferentes propostas.

	PBFT	Zyzyzya	MinBFT	Mencius	Steward	EBAWA	ZZ	SMIT	IVTM	IVTM Distribuído
Passos de Comunicação	4	2	3	3	2/4	3	4/2	2	2	4/2
Tipo de Ambiente	LAN	LAN	LAN	WAN	WAN	WAN	LAN	LAN	LAN	WAN
Tipo de Falhas	Bizantino	Bizantino	Bizantino	<i>Crash</i>	Híbrido	Bizantino	Bizantino	Bizantino	Bizantino	Bizantino
Número de Réplicas	3f+1	3f+1	2f+1	3f+1	2g+1/3f+1	3f+1	3g+1/f+1	2f+1	f+1	2g+1/f+1

Na Tabela 6.12 voltamos a mostrar as propostas já discutidas no capítulo 2 e confrontadas com os nossos resultados (IVTM, capítulo 4 e IVTM Distribuído, capítulo 5). Esta tabela evidencia os desempenhos em situações normais dos protocolos em termos de passos em um *round*. Também mostra o ambiente para o qual foram desenvolvidos e o tipo de falhas que tratam. Por fim, a tabela evidencia o número de réplicas necessárias nas ME para a tolerância a falhas.

## 6.4 CONCLUSÃO DO CAPÍTULO

Neste capítulo, foram apresentadas análises do protocolo hierárquico introduzido no capítulo 5. Também mostramos os nossos esforços de experimentações em um protótipo desenvolvido para verificar e evidenciar as características do modelo desenvolvido de ME em nossos estudos. Também confrontamos nossas propostas com a literatura relacionada.

O objetivo principal das análises, testes e considerações apresentadas neste capítulo foi de verificar o comportamento de nossa abordagem na presença de intrusões no sistema. Esperamos que as nossas conclusões neste capítulo possam ter mostrado nossos possíveis avanços e as limitações também de nossas propostas.

## 7 CONCLUSÃO

O presente capítulo conclui este documento de tese. Primeiramente, apresentamos uma visão geral sobre os trabalhos desenvolvidos. Na sequência, os objetivos desta tese são lembrados e as contribuições da mesma são abordadas, no sentido de examinar até que ponto os objetivos iniciais foram contemplados pelas nossas atividades neste período de doutoramento. Por fim, algumas perspectivas de trabalhos futuros são expostas.

### 7.1 VISÃO GERAL DO TRABALHO

Este trabalho apresentou os esforços no desenvolvimento de modelos e arquiteturas para serviços tolerantes a intrusões em ambientes de redes de longas distâncias. A nossa primeira experiência com ME visando a tolerância a intrusões foi descrita capítulo 4. Descrevemos neste capítulo o modelo ITVM que faz uso da virtualização para tolerar e tratar réplicas de serviço corrompidas por intrusões. A noção de elemento confiável é central neste modelo. Também neste modelo, é enfatizada a separação da execução da aplicação em réplicas da ME, do serviço de acordo necessário. O modelo de ME é todo alocado em uma máquina física servidora. A execução de um protocolo de consenso BFT fica muito simplificado quando executado sobre memória compartilhada. O custo do protocolo no caso otimista é de 2 passos de comunicação isto porque a maioria das etapas do acordo BFT são internos, executados em memória compartilhada. A proposta foi desenvolvida considerando intrusões e somente falhas maliciosas. O número de réplicas envolvidas na ME varia entre  $f + 1$  (caso otimista) e  $2f + 1$  (em presença de falhas).

Na literatura muitos trabalhos sobre faltas bizantinas apresentam o limite máximo de réplicas maliciosas fixo (em  $f$  falhas) que permite aos protocolos o progresso de sua execução numa primeira etapa. Mas, com as falhas ocorrendo no ciclo de vida destas propostas, reduzem a capacidade de tolerar intrusões e falhas com o tempo. Se não existe a recuperação automática dos elementos corrompidos (faltosos) a capacidade de tolerar estes comportamentos faltosos vai sendo reduzida com o tempo. Além disso, quando esse limite de  $f$  falhas é alcançado durante a execução do serviço, este deixa de funcionar corretamente. O modelo do ITVM, porque está fundamentada a sua replicação ME em virtualização, apresenta esta capacidade de tratamento de réplicas faltosas. Com isto, réplicas maliciosas são removidas, de modo que, o número de réplicas corretas sempre inicie uma execução de protocolo em configuração mínima ( $f + 1$  réplicas). E evolui em situação de desacordo para suportar um número ilimitado de faltas com a configuração máxima ( $2f + 1$ ). A ME é sempre restaurada na sua configuração mínima de  $f + 1$  réplicas no término do acordo. A vantagem do nosso sistema com o tratamento das réplicas maliciosas é que o número de intrusões é limitado por instâncias de protocolo, mas é ilimitada durante o ciclo de vida do servidor replicado.

Um protótipo do ITVM foi desenvolvido e testado para verificar a eficiência deste modelo. Também um estudo foi realizado sobre orquestração

distribuída. A ideia de usar composição de serviços para construir serviços replicados foi concretizada com o ITVM. Um serviço foi composto de camadas de orquestrações para controlar a execução do serviço ME implementado em nosso protótipo. Num primeiro nível tínhamos *engines* atribuídas para o controle de réplicas em suas VMs. A combinação destes *engines* formando uma ME era feita por outro nível de orquestração que combinava as diferentes *engines* de réplicas. A linguagem usada na descrição da especificações destas composições foi para descrever orquestrações através de um processo de negócio executável *BPEL* [Arkin, *et al.*, 2007]. Estes resultados de implementação estão no capítulo 4.

Os nossos esforços para estender o IVTM para ambientes distribuídos de rede de longa distância são descritos no capítulo 5. Desenvolvemos um protocolo hierárquico que sustenta a replicação ME do ITVM. Este protocolo foi desenvolvido com uma estrutura hierárquica visando redes de longa distância. A separação de faltas em modelo híbrido e a possibilidade de confinar os custos de BFT em sítios torna o protocolo adequado para ambientes de longa distância. Em nível global de sistemas distribuídos o protocolo define uma necessidade menor de replicação devido à limitação de falhas neste nível a somente comportamentos mais simples de crashes. O protocolo, neste nível, necessita de uma replicação reduzida se comparado com outros algoritmos de BFT. Neste nível global nos inspiramos no Paxos. A troca de visão global (troca de líder) é a única ação que realizamos neste nível. Estas trocas se efetuadas a cada  $f$  trocas de visão local em  $g + 1$  sítios. Neste nível não ocorre o tratamento dinâmico de elementos faltosos. Em nível global são atendidos os requisitos de disponibilidade. A versão estendida, chamamos de ITVM Distribuído. A diferença deste modelo, do descrito no capítulo 4, é que este último atende requisitos de disponibilidade. Ataques de *DoS* são suportados bem melhor que a versão anterior.

Os nossos trabalhos também contemplaram análises e testes que tiveram a finalidade de verificar propriedades e de testar as funcionalidades do protocolo hierárquico de ME. Um protótipo foi implementado concretizando este modelo de replicação para ambientes de longa distância. As nossas propostas foram também confrontadas neste capítulo com a literatura relacionada.

## 7.2 REVISÃO DOS OBJETIVOS

O objetivo geral era o desenvolvimento de modelos que, usando o conceito de elemento confiável, fizessem a separação de comportamentos faltosos. Com isto produziríamos um modelo híbrido de tolerância a intrusões e faltas, separando faltas bizantinas de *crashes*. As faltas bizantinas por serem mais caras teriam seus mecanismos restritos ao nível local (de máquina servidora). O tratamento destas faltas usaria replicação MEs implementadas com o uso da virtualização. As faltas de *crash* que envolvem menores custos seriam tratadas a nível de sistema distribuídos.

Acreditamos que este objetivo geral foi o foco principal que norteou nossos estudos durante este período de doutoramento.

Os objetivos específicos desta tese, enunciados na introdução, são aqui lembrados, seguidos de uma indicação dos estudos e resultados que visaram cumprir estes objetivos:

- **Estudos de algoritmos de BFT projetados para ambientes de larga escala que serviram de base para nossas proposições.**

O capítulo 2 é consequência deste objetivo específico.

- **Concretização de modelos de ME usando virtualização e concretizando os conceitos e propriedades de elemento confiável e de separação de faltas.**

Os trabalhos desenvolvidos principalmente no capítulo 4 cumpriram com estes objetivos.

- **Estratificação de soluções para ambientes de larga escala como *WANs* de modo a produzir protocolos hierárquicos menos custosos que propostas *flats*.**

O modelo ITVM foi desenvolvido visando reduzir os custos de propostas normalmente encontrada na literatura, Sítios concentram as replicações ME usadas na tolerância a intrusões. A redundância de sítios atende os requisitos de disponibilidade do serviço.

- **Uso de composição via várias *engines* para produzirem orquestrações descentralizadas e hierarquizadas de serviços replicados.**

Este objetivo foi atendido com o ITVM sem redundância física. Usamos o protótipo desenvolvido e estruturamos o mesmo como um serviço composto de orquestrações descentralizadas. O capítulo 4 descreve esta experiência.

Faltou definir um serviço como uma combinação de *engines* descentralizadas para ser usado com o suporte do protótipo do ITVM Distribuído.

- **Desenvolvimento de protótipos (infraestruturas) de nossos modelos de BTF que permitem de maneira flexível a construção de Máquinas de Estado tolerantes a intrusões, atendendo também requisitos de larga escala.**

Estes protótipos foram desenvolvidos para as duas experiências de modelos desenvolvidos nesta tese. O capítulo 4 e o capítulo 6 descrevem estes protótipos.

- **Produção de artigos.**

Foram produzidos dois artigos:

- SBRC: Lau, J.; BARRETO, L.; FRAGA, J. S. *Infraestrutura Baseada em Virtualização para Serviços Tolerantes a Intrusão*. In: XXX Simpósio

Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2012), 2012, Ouro Preto, MG. Anais do SBRC'2012, 2012. p. 45-54.

- ICWS: J.; BARRETO, L.; FRAGA, J. S. *An Infrastructure Based in Virtualization for Intrusion Tolerant Services*. In: 19th IEEE International Conference on Web Services (ICWS'2012), 2012, Honolulu, EUA. Proceedings of IEEE ICWS'2012. Los Alamitos: IEEE Computer Society, 2013. p. 170-177.

- Dois artigos estão sendo escritos para submissões em periódico científico e em simpósio internacional.

### 7.3 CONTRIBUIÇÕES DA TESE

Dentre as principais contribuições dos trabalhos desenvolvidos, está a introdução de modelos de tolerância a intrusões que possam ter a capacidade de permitir que clientes e provedores de serviços interajam de forma segura e disponível. Estes modelos descrevem aspectos como registro de nomes completamente descentralizado facilitando o acesso ao serviço replicado. A versão distribuída foi desenvolvida com conceitos visando confinar em sítios as partes mais custosas ligadas à BFT. As trocas entre sítios são projetadas para envolverem menos necessidade de réplicas e custos. Não diminuimos o número de passos na parte distribuída.

Os nossos trabalhos com orquestrações descentralizadas foram pioneiras no Brasil. Não conseguimos cumprir com nossos objetivos neste tópico em relação ao ITVM Distribuído porque o tempo de desenvolvimento do protocolo distribuído ocupou grande parte de nossos esforços.

Outro ponto que considero importante foram as soluções encontradas com a virtualização nos dois modelos. Também foram importantes os nossos esforços em testes e análises tentando mostrar nossos possíveis avanços e as limitações de nossas propostas.

### 7.4 TRABALHOS FUTUROS

Na verdade, não esgotamos todas as possibilidades de nossos modelos. A implementação do ITVM Distribuído não se mostrou muito eficiente. As limitações de tempo e mesmo a falta de experiência com a virtualização devem ter contribuído para nossos resultados não serem tão expressivos. Uma boa revisão de nossas implementações no protótipo do protocolo hierárquico teria que ser feita.

Outra parte não atendida no protótipo distribuído que construímos de nossas propostas, foi o requisito de escalabilidade. Seria muito importante envolver replicações com números mais significativos de sítios que o usado em nosso protótipo atual. As medidas que se conseguiríamos neste caso poderiam ser comparadas com outras propostas que visam atender este requisito como o sistema *Steward* [Amir, *et al.*, 2010] e o *EBWA* [Veronese, *et al.*, 2010].

É importante também salientar a necessidade ter o suporte ITPV encapsulado na forma de um serviço formado por uma composição distribuída. O tópico orquestração descentralizada era um dos nossos objetivos e não foi cumprido durante o nosso doutorado. Serviços replicados usando a tecnologia

Web Services, ficam mais adaptáveis às necessidades de aplicações em redes de longa distância.

Por fim, diríamos que os trabalhos realizados foram muitos e complexos e que esperamos que de alguma forma, tenham contribuído para melhorar a visão em termos de segurança e tolerância a intrusões em ambientes abertos como os formados por redes de longa distância.





## 8 Referências Bibliográficas

- Aghdaie, N. e Tamir, Y. (2002). Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support. *Proc. IEEE Intl. Conf. on Computer Communications and Networks*.
- Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-Rotaru, C., Olsen, J. e Zage, D. (2006). Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. *Proc. Int. Conf. on Dependable Systems and Networks*.
- Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-Rotaru, C., Olsen, J. e Zage, D. (2010). "Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks." *IEEE Transactions on Dependable and Secure Computing* 7(1): 80-93.
- Arkin, A. e Askary, S. (2004). WS-BPEL: Web Services Business Process Execution Language Version 2.0, OASIS Open, December.
- Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Liu, C. K., Thatte, S., Yendluri, P. e Yiu, A. (2007). Web services business process execution language version 2.0. wsbpel-specificationdraft-01, April, 2007. OASIS.
- Atkinson, B., Della-Libera, G., Hada, S., Hondo, M. e Hallam-Baker, P. (2004). Specification: Web Services Security (WS-Security). 24.
- Avizienis, A. e Kelly, J. (1984). "Fault Tolerance by Design Diversity: Concepts and Experiments." *Computer* 17(8): 67-80.
- Avizienis, A., Lapried, J.-C., Randell, B. e Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*.
- Bessani, A. N., Sousa, P., Correia, M., Neves, N. F. e Verissimo, P. (2007). Intrusion-Tolerant Protection for Critical Infrastructures. Technical Report 07-8, April 2007. DI/FCUL.
- Binder, W., Constantinescu, I. e Faltings, B. (2006). "Decentralized Orchestration of Composite Web Services." *Proc. of the IEEE International Conference on Web Services (ICWS'06)*: 869-876.
- Birman, K. P. (1996). *Building Secure and Reliable Network Applications*, Manning Greenwich.
- Boichat, R., Dutta, P., Frølund, S. e Guerraoui, R. (2003). "Deconstructing Paxos." *ACM SIGACT News* 34(1): 47-67.
- Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C. e Orchard, D. (2004). "Web Services Architecture." *W3C Working Group Note* 11: 2005-1.

- Cachin, C., Guerraoui, R. e Rodrigues, L. (2011). *Introduction to reliable and secure distributed programming*, Springer.
- Castro, M. e Liskov, B. (1999). Practical Byzantine Fault Tolerance. Proceedings of the third Symposium on Operating Systems Design and Implementation. New Orleans, Louisiana, United States, USENIX Association.
- Castro, M. e Liskov, B. (2002). "Practical Byzantine Fault Tolerance and Proactive Recovery." *ACM Transaction on Computer Systems (TOCS)* 20(4): 398-461.
- Chafle, G. B., Chandra, S., Mann, V. e Nanda, M. G. (2004). Decentralized Orchestration of Composite Web Services. Proceedings of the 13th international World Wide Web conference on Alternate track papers. New York, NY, USA, ACM: 134-143.
- Chandra, T. e Toueg, S. (1996). "Unreliable Failure Detectors for Reliable Distributed Systems." *Journal of the ACM* 43(2): 225-267.
- Charron-Bost, B. e Schiper, A. (2004). "Uniform Consensus is Harder than Consensus." *Journal of Algorithms* 51(1): 15-37.
- Chinnici, R., Moreau, J., Ryman, A. e Weerawarana, S. (2007). Web Services Description Language (WSDL) Version 2.0, June, 2007.
- Chun, B.-G., Maniatis, P., Shenker, S. e Kubiataowicz, J. (2007). "Attested Append-Only Memory: Making Adversaries Stick to their Word." *Proceeding of the 21st ACM Symposium on Reliable Distributed Systems* 41(6): 189-204.
- Chun, B., Maniatis, P. e Shenker, S. (2008). Diverse Replication for Single-Machine Byzantine-Fault Tolerance. *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, Boston, Massachusetts.
- Clement, L., Hatley, A., von Riegen, C. e Rogers, T. (2004). "UDDI Version 3.0. 2." *UDDI Spec Technical Committee Draft, Dated 20041019: 0.2-20041019*.
- Correia, M., Neves, N. e Verissimo, P. (2005). "From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures." *The Computer Journal* 49(1): 82.
- Correia, M., Verissimo, P. e Neves, N. F. (2002). The Design of a COTS Real-Time Distributed Security Kernel. Proceedings of the European Dependable Computing Conference (EDCC'04), Springer Berlin / Heidelberg: 234-252.
- Correia, M., Veronese, G. S. e Lung, L. C. (2010). Asynchronous Byzantine Consensus with  $2f+1$  Processes. *Proceedings of the 25th Annual ACM Symposium on Applied Computing (SAC'10)*.
- Correia, M. P. (2005). Serviços Distribuídos Tolerantes a Intrusões: Resultados Recentes e Problemas Abertos, Faculdade de Ciências da Universidade de Lisboa.
- Cristian, F. (1991). "Understanding fault-tolerant distributed systems." *Communications of the ACM* 34(2): 56-78.

- Deswarte, Y., Kanoun, K. e Laprie, J. C. (1998). Diversity Against Accidental and Deliberate Faults. *Proceedings Computer Security, Dependability and Assurance: From Needs to Solutions*.
- Dialani, V., Miles, S., Moreau, L., Roure, D. D. e Luck, M. (2002). Transparent Fault Tolerance for Web Services based Architectures *Eighth International Europar Conference (EURO-PAR'02)*.
- Dobson, G. (2006). Using WS-BPEL to Implement Software Fault Tolerance for Web Services. *Proc. of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06)*.
- Doudou, A., Garbinato, B., Guerraoui, R. e Schiper, A. (1999). "Muteness failure detectors: Specification and implementation." *European Dependable Computing Conference (EDCC'99)* 1667: 71-87.
- Doudou, A. e Schiper, A. (1997). Muteness Detectors for Consensus with Byzantine Processes. *17th ACM symposium on Principles of distributed computing (PODC '98)*.
- Dwork, C., Lynch, N. e Stockmeyer, L. (1988). "Consensus in the Presence of Partial Synchrony." *Journal of the ACM (JACM)* 35(2): 288-323.
- Fernandez, A., Jimenez, E. e Raynal, M. (2007). Electing an Eventual Leader in an Asynchronous Shared Memory System. *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*.
- Fraga, J. d. S. e Powell, D. (1985). A Fault- and Intrusion-Tolerant File System. *Proceedings of the 3rd International Conference on Computer Security*.
- Gafni, E. e Lamport, L. (2003). "Disk paxos." *Distributed Computing* 16(1): 1-20.
- Garcia, M., Bessani, A. e Neves, N. (2011). Diverse OS Rejuvenation for Intrusion Tolerance. *International Conference on Dependable Systems and Networks (DSN'11)*, Hong Kong, China.
- Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J. J. e Nielsen, H. F. (2003). SOAP Version 1.2 Part 1: Messaging Framework, June.
- Hadzilacos, V. e Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. New York-USA, Technical report, Department of Computer Science, Cornell University.
- Júnior, V. S., Lung, L. C., Correia, M., Fraga, J. d. S. e Lau, J. (2010). Intrusion Tolerant Services Through Virtualization: a Shared Memory Approach. *24th IEEE International Conference on Advanced Information Networking and Applications (AINA'10)*, Perth, Australia.
- Kavantzias, N., Burdett, D., Ritzinger, G., Fletcher, T. e Lafon, Y. (2005). "Web Services Choreography Description Language Version 1.0." *W3C Working Draft* 17: 10-20041217.

- Kirsch, J. e Amir, Y. (2008). Paxos for System Builders, Technical Report CNDS-2008-2, Johns Hopkins University.
- Kotla, R., Alvisi, L., Dahlin, M., Clement, A. e Wong, E. (2007). Zyzzyva: Speculative Byzantine Fault Tolerance. *Proc. of 20th ACM SIGOPS Symposium on Operating Systems Principles*, New York, NY, USA, ACM Press
- Kyprianou, N. (2008). Hybrid Web Services Orchestration. Computer Science, University of Edinbutgh.
- Lamport, L. (1998). "The Part-Time Parliament." *ACM Transactions on Computer Systems (TOCS)* 16(2): 133-169.
- Lamport, L. (2001). "Paxos made simple." *ACM SIGACT News* 32(4): 18-25.
- Lamport, L. (2006). "Fast Paxos." *Distributed Computing* 19(2): 79-103.
- Lamport, L. e Massa, M. (2004). Cheap paxos. *International Conference on Dependable Systems and Networks*, IEEE Computer Society.
- Lamport, L., Shostak, R. e Pease, M. (1982). "The Byzantine Generals Problem." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4(3): 382-401.
- Laprie, J. C. (1995). Dependable Computing and Fault Tolerance: Concepts and Terminology. *IEEE Proceedings of FTCS*.
- Laranjeiro, N. e Marco Vieira (2007). Towards Fault Tolerance in Web Services Compositions. *Proc. of the Workshop on Engineering Fault Tolerant Systems (EFTS'07)*.
- Lau, J., Lung, L. C., Fraga, J. d. S. e Veronese, G. S. (2008). Designing Fault Tolerant Web Services Using BPEL. *Proceeding of 7th IEEE/ACIS International Conference on Computer and Information Science (ICIS'08)*, Poartland, USA
- Liang, D., Fang, C.-L., Chen, C. e Lin, F. (2003). *FT-SOAP: A Fault-tolerant web service*.
- Littlewood, B. e Strigini, L. (2004). "Redundancy and Diversity in Security." *Lecture Notes in Computer Science* 3193: 423-438.
- Lynch, N. (1996). *Distributed Algorithms*, Morgan Kaufmann.
- Mao, Y., Junqueira, F. P. e Marzullo, K. (2008). Mencius: Building Efficient Replicated State Machines for WANs. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. San Diego, California, USENIX Association: 369-384.
- Marandi, P. J., Primi, M., Schiper, N. e Pedone, F. (2010). Ring Paxos: A High-Throughput Atomic Broadcast Protocol. *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10)*, Chicago, IL

- Martin, J. e Alvisi, L. (2006). "Fast Byzantine Consensus." *IEEE Transactions on Dependable and Secure Computing*: 202-215.
- Mendling, J. e Hafner, M. (2008). "From WS-CDL choreography to BPEL process orchestration." *Journal of Enterprise Information Management* 21(5): 525–542.
- Merideth, M. G., Iyengar, A., Mikalsen, T., Tai, S., Rouvellou, I. e Narasimhan, P. (2005). Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications. *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*.
- Milanovic, N. e Malek, M. (2004). "Current solutions for Web service composition." *Internet Computing, IEEE* 8(6): 51-59.
- Moses, T. (2005). "eXtensible Access Control Markup Language (XACML) version 2.0." *Oasis Standard* 200502.
- Murthy, C. S. R. e Manoj, B. (2004). *Ad hoc wireless networks: Architectures and protocols*, Pearson education.
- Nadalin, A., Kaler, C., Monzillo, R. e Hallam-Baker, P. (2005). Web Services Trust Language (WS-Trust) V1.1. OASIS. 7.
- Nanda, M. G., Chandra, S. e Sarkar, V. (2004). Decentralizing Execution of Composite Web Services. Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04). Vancouver, BC, Canada, ACM: 170-187.
- Neves, N. F., Correia, M. e Verissimo, P. (2004). Wormhole-Aware Byzantine Protocols. *2nd Bertinoro Workshop on Future Directions in Distributed Computing: Survivability - Obstacles and Solutions (FuDiCo: SOS)*, Bertinoro, Italy.
- Newsome, J., Shi, E., Song, D. e Perrig, A. (2004). The sybil attack in sensor networks: analysis & defenses. *Proceedings of the 3rd international symposium on Information processing in sensor networks*, ACM.
- OASIS (2005). OASIS - Organization for the Advancement of Structured Information Standards. , [www.oasis.open.org](http://www.oasis.open.org).
- Obelheiro, R. R., Bessani, A. N., Lung, L. C. e Correia, M. (2006). How Practical are Intrusion-Tolerant Distributed Systems. Technical Report. Department of Informatics, University of Lisbon Technical Report 06-15.
- OGSA (2003). Open Grid Services Architecture, [www.globus.org/ogsa](http://www.globus.org/ogsa).
- OMG (2000). Fault-Tolerant in CORBA Specification v.1.0, OMG Document ptc/2000-04-04.
- Peltz, C. (2003). Web Services Orchestration - a review of emerging technologies, tools, and standards. Relatório Técnico H. P. Company.

- Pullum, L. L. (1993). A new adjudicator for fault tolerant software applications correctly resulting in multiple solutions. *IEEE Digital Avionics Systems Conference (DASC'93)*.
- Ragouzis, N., Hughes, J., Philpott, R. e Maler, E. (2005). "Security Assertion Markup Language (SAML) V2. 0." *OASIS*.
- Raynal, M. e Singhal, M. (2001). "Mastering Agreement Problems in Distributed Systems." *IEEE Software* 18(4): 40-47.
- Reiser, H. e Kapitzka, R. (2009). "Towards Recoverable Hybrid Byzantine Consensus."
- Reiser, H. P., Hauck, F. J., Kapitzka, R. e Schroder-Preikschat, W. (2006). Hypervisor-based redundant execution on a single physical host. *Proc. of the 6th European Dependable Computing Conference (EDCC'06)*, Coimbra, Portugal.
- Reiser, H. P. e Kapitzka, R. (2007). "VM-FIT: Supporting Intrusion Tolerance with Virtualisation Technology." *Proceedings of the Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS'07)*.
- Reynal, M. (2005). "A Short Introduction To Failure Detectors For Asynchronous Distributed Systems." *ACM SIGACT News* 36(1): 53-70.
- Ross-Talbot, S. e Fletcher, T. (2006). Web services choreography description language: Primer (working draft), Technical report, W3C, June 2006: <http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/>.
- Sahoo, J., Mohapatra, S. e Lath, R. (2010). Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. *International Conference on Computer and Network Technology (ICCNT'10)*.
- Salas, J., Perez-Sorrosal, F., Patiño-Martínez, M. e Jiménez-Peris, R. (2006). "WS-Replication: A Framework for Highly Available Web Services." *Proceedings of the 15th international conference on World Wide Web*: 357-366.
- Santos, G. T., Lung, L. C. e Montez, C. (2005). FTWeb: A Fault Tolerant Infrastructure for Web Services. *Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)*.
- Schneider, F. B. (1990). "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial." *ACM Computing Surveys* 22(4): 299-319.
- Sousa, P., Bessani, A. e Obelheiro, R. R. (2008). The FOREVER Service for Fault/Intrusion Removal. *Proceedings of the 2nd Workshop on Recent advances on Intrusion-Tolerant Systems (WRAITS 2008)*, Glasgow, UK, ACM New York, NY, USA.
- Stoess, J. L. V. U. J. e Götz, S. (2004). Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, San Francisco, CA.

- Tai, S., Khalaf, R. e Mikalsen, T. (2004). "Composition of coordinated web services." *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*: 294-310.
- Tartanoglu, F., Issamy, V., Romanovsky, A. e Levy, N. (2003). "Coordinated Forward Error Recovery for Composite Web Services." *Proceeding of the 22nd International Symposium on Reliable Dependable Systems (SRDS'03)*.
- Townend, P. e Xu, J. (2005). Fault Tolerance within a Grid Environment. *Engineering and Physical Sciences Research Council (EPSRC'05)*.
- Verissimo, P. e Rodrigues, L. (2001). *Distributed systems for system architects*, Springer Netherlands.
- Verissimo, P. E., Neves, N. F. e Correia, M. P. (2003). Intrusion-Tolerant Architectures: Concepts and Design. Architecting Dependable Systems R. Lemos, C. Gacek e A. Romanovsky. Springer-Verlag, Lecture Notes in Computer Science: 3-36.
- Veronese, G., Correia, M., Bessani, A. e Lung, L. (2009). Highly-Resilient Services for Critical Infrastructures. *In Proceedings of the Embedded Systems and Communications Security workshop (ESCS '09)*.
- Veronese, G., Correia, M., Bessani, A. e Lung, L. (2009). Spin one's wheels? Byzantine fault tolerance with a spinning primary. D.-F.-. Technical Reports.
- Veronese, G. S., Correia, M., Bessani, A. N. e Lung, L. C. (2010). EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. *Proceedings of the 12th IEEE International High Assurance Systems Engineering Symposium (HASE'10)*.
- Veronese, G. S., Correia, M., Lung, L. C., Bessani, A. N. e Verissimo, P. (2008). Minimal Byzantine Fault Tolerance: Algorithms and Evaluation. TR-2009-15, June 2009. DI/FCUL
- Vogels, W. (2003). "Web Services Are Not Distributed Objects." *Internet Computing* 7(6): 59-66.
- W3C (2005). W3C (World Wide Web Consortium). [www.w3c.org](http://www.w3c.org).
- Wood, T., Singh, R., Venkataramani, A., Shenoy, P. e Cecchet, E. (2011). ZZ and the Art of Practical BFT Execution. *Extended Technical Report for EuroSys 2011*, ACM.
- WS-RM (2004). Web Services Reliable Messaging <ftp://www6.software.ibm.com/software/developer/library/wsreliablemessaging200502.pdf>.
- XEN (2009). How Does Xen Work. <http://xen.org/files/Marketing/HowDoesXenWork.pdf>.

- Yin, J., Martin, J., Venkataramani, A., Alvisi, L. e Dahlin, M. (2003). "Separating Agreement from Execution for Byzantine Fault Tolerant Services." *ACM SIGOPS Operating Systems Review* 37(5): 253-267.
- Zdun, U., Hentrich, C. e Van Der Aalst, W. (2006). "A Survey of Patterns for Service-Oriented Architectures." *International Journal of Internet Protocol Technology* 1(3): 132-143.
- Zhang, X., Zagorodnov, D., Hiltunen, M., Marzullo, K. e Schlichting, R. D. (2004). "Fault-Tolerant Grid Services Using Primary-Backup: Feasibility and Performance." *IEEE International Conference on Cluster Computing (ICCC'04)*: 105-114.
- Zhao, W. (2007). "BFT-WS: A Byzantine Fault Tolerance Framework for Web Services." *Proc. Middleware for Web Services Workshop*.
- Zhao, W. (2008). "Design and Implementation of a Byzantine Fault Tolerance Framework for Web Services." *The Journal of Systems & Software*.