Universidade Federal de Campina Grande
Departamento de Sistemas e Computação
Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

Projeto e Implementação de um Serviço de Detecção de Falhas com Semântica Perfeita para Redes Locais

Ely Wagner Aguiar de Oliveira
ely@dsc.ufcg.edu.br

Orientador:
Francisco Vilar Brasileiro
fubica@dsc.ufcg.edu.br

Campina Grande, Agosto de 2003

OLIVEIRA, Ely Wagner Aguiar de

O48P

Projeto e Implementação de um Serviço de Detecção de Falhas com Semântica Perfeita para Redes Locais

Dissertação de Mestrado, Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática,

Campina Grande, Paraíba, Agosto de 2003.

134 p. Il.

Orientador: Francisco Vilar Brasileiro

Palavras Chave:

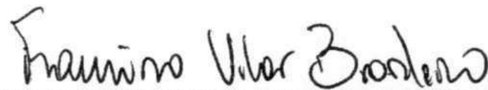
1. Tolerância à Falhas
2. Detectores de Falhas não Contáveis
3. Semântica Perfeita
4. Serviço Linux

CDU - 681.3.066D

"PROJETO E IMPLEMENTAÇÃO DE UM SERVIÇO DE DETECÇÃO DE FALHAS COM SEMÂNTICA PERFEITA PARA REDES LOCAIS"

ELY WAGNER AGUIAR DE OLIVEIRA

DISSERTAÇÃO APROVADA EM 29.08.2003



PROF. FRANCISCO VILAR BRASILEIRO, Ph.D
Orientador

PROF. JACQUES PHILIPPE SAUVÉ, Ph.D
Examinador

PROFª. FABÍOLA GONÇALVES P. GREVE, Dra.
Examinadora

CAMPINA GRANDE – PB

Resumo

Na intenção de ampliar seus serviços, a cada dia que passa, diversos setores de nossa sociedade, como bancos, hospitais e indústrias, aumentam ainda mais seu grau de dependência do correto funcionamento de sistemas distribuídos. Em paralelo a esta realidade, está o fato de que, como qualquer ambiente computacional, os sistemas distribuídos estão sujeitos a ocorrência de falhas, que se não forem tratadas adequadamente, podem comprometer a realização de suas tarefas. Isto lança um desafio a seus projetistas e desenvolvedores, que é o de atender à crescente demanda por confiabilidade em sistemas distribuídos, cada vez mais expostos a situações de falha.

Já que as falhas não podem ser evitadas totalmente, os sistemas devem se valer de mecanismos que os permitam tolerá-las, detectando-as e tratando-as, sem interromper seu funcionamento.

Um detector de falhas não confiável é uma importante abstração para viabilizar a implementação de protocolos tolerantes a falhas em sistemas distribuídos assíncronos. Algumas classes de detectores de falhas foram propostas, dentre as quais, a dos perfeitos, que possui a semântica mais forte. Neste trabalho é apresentado o projeto e a implementação do Delphus, um serviço de detecção de falhas com semântica perfeita, com garantias de qualidade de serviço. O serviço é implementado no nível do sistema operacional como um módulo para versões genéricas do Linux. O acesso ao serviço é disponibilizado através de APIs implementadas em C e Java. Um canal extra de comunicação interliga as máquinas onde o serviço executa, e é utilizado exclusivamente para o tráfego de suas mensagens.

O Delphus apresenta-se como uma importante ferramenta para a implementação de mecanismos de tolerância a falhas, a um custo baixo e sem exigir a adoção de grandes restrições no sistema.

Abstract

In the intention to extend its services, several sectors of our society, as banks, hospitals and industries, increase even more its degree of dependence of the correct functioning of distributed systems. In parallel to this reality, is the fact that, as any other computational environment, distributed systems are exposed to the occurrence of faults. If not properly treated, these faults can prevent the distributed system of completing its tasks. It challenges its designers and developers, to handle the increasing demand for dependability in distributed systems, even more exposed to the occurrence of faults. Since faults can not be totaly prevented, systems must use fault tolerance mechanisms. Such mechanisms allow faults to be detected and treated, without interrupting system functioning.

Unreliable failure detectors are an important abstraction to suport the implementation of fault tolerant protocols on asynchronous distributed systems. Several classes of failure detectors with varying semantics have been proposed, such as the class of Perfect Failure Detectors, which is the strongest one.

This work presents the design and implementation of Delphus, a perfect failure detection service with quality of service. It is a new operational system service, implemented as a module for Linux generic versions. The access to the service is provided by APIs implemented in both C and Java. An extra communication channel is used solely to convey service messages.

The Delphus is presented as a cheap and important tool to support the implementation of fault tolerance mechanisms, without demanding the adoption of strong restrictions in the system.

Agradecimentos

A meus amigos conquistenses Gil, Quezia, Nilzane e tia Gislane, por terem sido minha referência de família no tempo que passei na Paraíba.

A meu amigo Alex que mesmo a distância, em todo o tempo que passei em Campina, sempre esteve tão próximo a mim quanto quando ainda morava em Salvador.

A meus amigos da UFCG, especialmente aos colegas do LSD e mais especialmente ainda a Lauro, Andrey e Nazareno, por terem sido amigos formidáveis.

Ao meu orientador Fubica, por todo apoio e incentivo recebidos e pelo interesse demonstrado em meu sucesso.

À Agência Nacional do Petróleo, pelo suporte financeiro ao meu projeto de pesquisa, fundamental para sua realização.

À minha irmãzinha Zama, por compartilharmos cada momento de entusiasmo, tristeza, expectativa e alegria que vivemos.

A meus pais, Cida e Walter, sem os quais este trabalho não poderia sequer ter sido sonhado. Eu o dedico a vocês por todo o sacrifício realizado em favor da minha felicidade. O amor e a educação que vocês me deram serão sempre ingredientes fundamentais de todas as minhas realizações.

Enfim, a Deus, pelas oportunidades que tive na vida e por sempre contar com a companhia de um amigo, aonde quer que eu vá.

OLIVEIRA, Ely Wagner Aguiar de O48M

Projeto e Implementação de um Serviço de Detecção de Falhas com Semântica Perfeita para Redes Locais

Dissertação de Mestrado, Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Paraíba, Agosto de 2003.

134 p. Il.

Orientador: Francisco Vilar Brasileiro

Palavras Chave: 1. Tolerância à Falhas 2. Detectores de Falhas não Confiáveis 3. Semântica Perfeita 4. Serviço Linux

CDU - 681.3.066D

Conteúdo

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivo 5

1.3 Contribuições do Trabalho 5

1.4 Organização do Trabalho 6

2 Detecção de Falhas em Sistemas Distribuídos **7**

2.1 Modelos de Falhas 7

2.2 Modelos de Sistemas 8

2.2.1 Modelo Síncrono 8

2.2.2 Modelo Puramente Assíncrono 9

2.3 Detectores de Falhas Nao Confiáveis 9

2.4 Aspectos de Qualidade de Serviço 13

2.5 Implementações de Detectores de Falhas 14

2.5.1 ESPRIT OpenDREAMS 16

2.5.2 Monitor 18

2.5.3 Licenças e Watchdogs 20

2.5.4 TCB (Timely Computing Base) 21

3 Projeto do Serviço Delphus **24**

3.1 Modelos de Sistema e de Falhas 24

3.2 Arquitetura do Serviço 25

3.3 Descrição do Serviço 27

3.3.1 Eleição de Líder 29

3.3.2 Admissão de Monitoráveis 30

3.3.3 Escalonamento de Transmissão de Heartbeats 33

3.4 API 37

3.4.1 Administração do Serviço 38

3.4.2 Configuração de Monitoração 38

3.4.3 Notificação de Falha 39

4 Implementação do Serviço Delphus 40

4.1 Descrição do Hardware 42

4.2 Descrição do Software 44

4.2.1 Threads do Núcleo 44

4.2.2 Iniciação do Serviço 45

4.2.3 Execução do Serviço 47

4.2.4 Cálculo do Tamanho do Período TDMA 53

4.2.5 Consulta do Estado dos Monitoráveis 73

4.2.6 Formato das Mensagens 73

4.2.7 Notificação de Eventos 80

4.2.8 Condições de Corrida 82

4.3 API 84

4.3.1 API C 84

4.3.2 API Java 90

4.3.3 Ferramenta de Administração Olen 99

4.3.4 Comandos de Console 100

5 Usando o Serviço 104

6 Avaliação de Desempenho do Serviço 109

6.1 Total de memória utilizada pelo serviço 110

6.2 Percentual de ocupação da CPU pelo serviço 113

6.3 Sobrecarga da rede 114

6.4 Teste de desempenho geral do sistema 114

7 Conclusões 117

7.1 Análise do Trabalho 117

7.2 Considerações Finais 117

7.3 Trabalhos Futuros 118

Lista de Figuras

1 Reducibilidade de Classes de Detectores de Falhas [Fonte: Chandra e Toueg [16]] 11

2 Níveis de Sincronia em Sistemas Assíncronos 12

3 Arquitetura do ESPRIT OpenDREAMS [Fonte: Felber *et al.* [27]] 16

4 Monitor em Funcionamento 19

5 Arquitetura do TCB [Fonte: Veríssimo e Cassimiro [45]] 22

6 Modelo do Sistema 24

7 Camadas da Arquitetura do Delphus 25

8 Arquitetura do Delphus 26

9 Protocolo de Comunicação TDMA 27

10 Protocolo de Comunicação TDMA do Delphus 28

11 Falha do Líder do Domínio de Detecção 30

12 Eleição de Líder do Domínio de Detecção 30

13 Admissão de uma nova máquina 32

14 Sequência de Transmissões de *Heartbeats* 34

15 Super Períodos 34

16 Ajuste de Frequência de Transmissões 35

17 Threads de Núcleo 44

18 Composição de uma Fatia TDMA 55

19 Início do período do líder e sincronização de período nos demais módulos . . 60

20 Composição de τ 61

21 Otimização do cálculo de τ 62

22 Erro de sincronização dos períodos 63

23 Formato da Mensagem de Sincronização 74

24 Formato da Mensagem de Proposta de Liderança 75

25 Formato da Mensagem de Anúncio de Admissão de Máquina Monitorável . . 75

26 Formato da Mensagem de Anúncio de Admissão de Processo Monitorável . . 76

27 Formato da Mensagem de Anúncio de Remoção de Monitorável 77

28 Formato da Mensagem de Heartbeat 77

29 Formato da Mensagem de Requisição de Admissão de Máquina Monitorável . 78
30 Formato da Mensagem de Requisição de Admissão de Processo Monitorável . 78
31 Formato da Mensagem de Requisição de Remoção de Monitorável 79
32 Formato da Mensagem de Recusa de Admissão de Monitorável 79
33 Formato da Mensagem de Atualização da Lista de Monitoráveis 80
34 Formato da Mensagem de Compromisso 80
35 Estrutura das APIs 84
36 Objetos Monitoráveis 92
37 Objetos Notificáveis 93
38 Objetos principais do serviço 94
39 Exceções da API Java 98
40 Olen - Ferramenta de Administração de Domínio de Detecção 100
41 Script *activate_backup* de configuração do servidor primário 105
42 Iniciação do serviço na máquina m1 106
43 Serviço iniciado nas 2 máquinas 106
44 Cadastro da kernel thread eth1 como monitorável 107
45 Cadastro do script activate_backup 107
46 Script activate_backup cadastrado 108
47 Consumo Total de Memória - Monitoráveis Locais 111
48 Consumo Total de Memória - Monitoráveis Globais 111
49 Consumo Total de Memória - Notificáveis 112
50 Percentual de Ocupação da CPU em 3 horas 113
51 Benchmark do Sistema com o Delphus 116

Lista de Tabelas

1 Classes de Detectores de Falhas 11

2 Medição de T_{com} e do total de interrupções 68

3 Medição de T_{prep}, T_{rec} e T_{trat} 69

4 Medição do total de interrupções 69

5 Medição de T_{prep}, T_{com}, T_{rec} e T_{trat} em máquina mais moderna 71

6 Medição do total de interrupções em máquina mais moderna 71

7 Relação entre $T_{periodo}$ e configurações do sistema 72

8 Códigos de Erro da API C 85

9 Exceções da API Java 99

10 Testes que compõem o Unixbench 115

1 Introdução

1.1 Motivação

Sistemas distribuídos são definidos como um conjunto de processos que executam em máquinas interligadas em rede e que realizam tarefas de forma cooperada. Hoje em dia, estes sistemas podem ser encontrados em diversos setores de nossa sociedade. Bancos, hospitais, comércio eletrônico, bolsas de valores, indústrias são apenas alguns exemplos. Na intenção de ampliar seus serviços, a cada dia que passa, estes setores aumentam ainda mais seu grau de dependência do correto funcionamento de tais sistemas.

Em paralelo a esta realidade, está o fato de que, como em qualquer ambiente computacional, falhas podem ocorrer em um sistema distribuído, e se não forem detectadas e tratadas adequadamente, a realização de suas tarefas poderá ser comprometida. Na verdade, devido à sua complexidade e heterogeneidade características, este tipo de sistema está bem mais sujeito à ocorrência de falhas, que os sistemas tradicionais centralizados. Isto lança um desafio a seus projetistas e desenvolvedores, que é o de atender à crescente demanda por confiabilidade em sistemas distribuídos, cada vez mais expostos a situações de falha.

Já que as falhas não podem ser evitadas totalmente, os sistemas devem se valer de mecanismos que lhes permitam detectá-las e tratá-las, sem interromper seu funcionamento. Eles precisam utilizar mecanismos de tolerância a falhas.

Algumas abstrações foram definidas para simplificar o projeto e implementação de sistemas distribuídos tolerantes a falhas. Dentre elas estão o Consenso e o Broadcast Atômico. Os algoritmos de Consenso permitem que vários processos do sistema distribuído atinjam decisões em comum. Já os de Broadcast Atômico permitem que mensagens sejam difundidas para um grupo de processos e que todos os processos operacionais recebam todas elas numa mesma ordem. Estes algoritmos devem contemplar a possibilidade de ocorrência de falhas nos processos ou no canal de comunicação. A base para o funcionamento desses algoritmos está na possibilidade de se detectar se um determinado processo falhou ou não. Por sua vez, essa detecção depende do nível de sincronismo existente no sistema.

Nos sistemas síncronos, onde é conhecido o limite máximo de tempo para transmissão de

mensagens entre os processos, a detecção de processos falhos pode limitar-se ao uso de *timeouts* (limites de tempo de comunicação), considerando falhos os processos que não enviam uma mensagem até o limite máximo de tempo estabelecido. Já em sistemas assíncronos, onde tais limites não existem, outras técnicas se fazem necessárias, uma vez que o recebimento de mensagens pode ser atrasado não por falhas, mas por sobrecarga na rede ou lentidão nos processos. Sendo assim, simplesmente basear-se em *timeouts* pode levar ao mau funcionamento do sistema como um todo, uma vez que processos corretos podem ser interpretados como falhos.

O modelo de sistema assíncrono traz algumas vantagens sobre o síncrono. Ele possui uma semântica simples e permite a construção de aplicações mais portáteis que as que se baseiam na existência de limites de tempo na execução de tarefas. Outra vantagem é o fato de apresentarem maior escalabilidade. Para garantir limites de tempo, os sistemas síncronos possuem limites bem definidos de carga de trabalho e número de componentes (máquinas, processos) que podem suportar. Nos sistemas assíncronos, esses limites virtualmente não existem, o que permite que possam ser ampliados, de acordo com o crescimento da demanda, sem que comprometam o funcionamento de suas aplicações. Além do mais, os sistemas síncronos, em sua maioria, adotam tecnologias proprietárias, o que restringe os recursos disponíveis para sua expansão, provocando sua dependência de uma tecnologia específica. Por esses motivos, grande é o número de aplicações desenvolvidas para o modelo de sistema assíncrono.

Entretanto, o resultado apresentado por Fischer *et al.* [30] prova ser impossível, em um sistema puramente assíncrono, sujeito a falhas, criar um protocolo que garanta, de forma determinista, que o consenso seja obtido. A razão está no fato de que, devido às incertezas no tempo de envio e entrega das mensagens, é impossível distinguir se um processo não responde por ter falhado ou por estar apenas muito lento. Esta dificuldade tem motivado muitas pesquisas e levado à criação de novos modelos de sistemas e à criação de protocolos de consenso para esses modelos, que tentam contornar tal impossibilidade.

Um dos novos modelos que ultimamente tem recebido mais atenção na literatura, é o do sistema assíncrono equipado com um detector de falhas não confiável. A idéia básica dos detectores de falhas não confiáveis, introduzida por Chandra e Toueg [16], é incorporar

ao sistema um serviço que forneça informações sobre os componentes suspeitos de terem falhado. Essas informações nem sempre são confiáveis, pois o detector pode cometer erros, seja suspeitando dos componentes corretos ou não suspeitando dos que realmente falharam.

A semântica do serviço de detecção é caracterizada através da definição de duas propriedades básicas: i) abrangência, que determina o mínimo de processos corretos que deverão suspeitar de um processo que falha; e ii) exatidão, que limita as falsas suspeições sobre processos que não falharam. Graduando os níveis de abrangência e exatidão, Chandra e Toueg criaram oito classes de detectores [16].

A classe $\Diamond S$ é considerada a de semântica mais fraca capaz de permitir uma solução determinista para o problema do consenso [17]. Por exigir o mínimo de restrições adicionais ao ambiente de execução, esta classe obteve um foco maior das pesquisas, favorecendo o surgimento de várias implementações de detectores de falhas desta classe [39, 19, 36, 10].

Apesar da simplicidade de implementação de detectores de semântica mais fraca, os algoritmos que os utilizam precisam considerar que eles podem cometer erros. Isso aumenta a complexidade de tais algoritmos, tornando seu desempenho provavelmente inferior ao que seria se o detector não cometesse erros. Em contrapartida, os detectores da classe P não cometem erros. Todo processo que falha, a partir de algum instante, é permanentemente detectado como falho, por todos os demais processos. Essa exatidão pode levar a um melhor desempenho na solução de consenso, que o observado com o uso de detectores de outras classes. Entretanto, para que não cometa erros, o detector necessita de um mínimo de sincronismo no sistema, que se traduz pela adoção de restrições, como por exemplo, a instalação de um segundo canal de comunicação dedicado à troca de mensagens do serviço de detecção, ou alterações no sistema operacional. Tais restrições fizeram com que a classe P recebesse menos atenção que as demais. Por isso, não existem muitas implementações dos mesmos.

Outro problema na implementação de detectores de falha é que, mesmo os da classe P, são especificados em termos de um comportamento de tempo incerto (após um tempo não limitado, o detector apresentará um certo comportamento)[1]. Entretanto, mesmo em sistemas assíncronos, aplicações frequentemente possuem restrições de tempo, ainda que essas restri-

[1]Em inglês, *eventually*, que significa que em algum momento, algo certamente ocorrerá, mesmo que sem a indicação de quando exatamente.

ções não sejam rígidas. Neste caso detectores de falhas que apresentam um comportamento "eventual" não são úteis. Um detector de falhas que suspeita de um processo que falhou após horas, certamente não será útil em um sistema que realiza diversos consensos por segundo. Portanto, aplicações podem necessitar de um detector de falhas que possua uma qualidade de serviço que possa ser quantificada [18].

Ademais, várias aplicações distribuídas demandam serviços perfeitos de detecção de falhas, com garantias de qualidade de serviço, para que consigam oferecer o nível de confiabilidade e desempenho que lhes é exigido.

O desempenho é um fator crítico em aplicações financeiras, como as que controlam operações em uma bolsa de valores. Muitas delas utilizam serviços como o de *Group Membership*. Ele permite que os processos se organizem em grupos e que as mensagens trocadas entre eles sejam recebidas em uma mesma ordem por todos os membros do grupo. Esta ordenação é um fator crítico, pois envolve a sequência de operações de compra e venda de ações e grandes recursos financeiros. Quando um processo é detectado como falho, ele é retirado do grupo. Se a suspeita de falha for falsa, ele é novamente incluído no grupo. Portanto, se um serviço de detecção de falhas que comete falsas suspeições, for utilizado, processos serão retirados e novamente incluídos nos grupos desnecessariamente, podendo reduzir o desempenho da aplicação e afetar as operações de compra e venda. Neste caso, é desejável um serviço que não cometa falhas, e que por outro lado, seja capaz de detectar uma falha dentro de um espaço de tempo limite. Um processo falho não detectado como tal pode atrasar a obtenção de consenso entre os processos, como a finalização de uma operação financeira, e levar um investidor a ter um grande prejuízo.

Não apenas a aplicações financeiras da bolsa de valores interessa o desempenho proporcionado pelo uso de um serviço perfeito de detecção de falhas, com garantias de qualidade de serviço. Em vários exemplos, como aplicações industriais e de comércio eletrônico, o desempenho é amplamente desejado, quando não um fator crítico.

Além do mais, somente um serviço de detecção de falhas com semântica perfeita permite a obtenção de consenso, independentemente do número de processos falhos. A semântica da classe $\diamond S$ permite a obtenção de consenso, desde que a maioria dos processos não falhe [17]. A probabilidade de mais que a metade dos processos falhar não pode ser desprezada em

sistemas de alto grau de confiabilidade e disponibilidade como os de monitoramento de tráfego aéreo ou ferroviário.

As poucas implementações de detectores de falha perfeitos existentes não atendem a um bom número destas aplicações por exigirem fortes restrições, que não podem ser adotadas em seus sistemas, como forte acoplamento com aplicações clientes e adoção de versões específicas de sistemas operacionais. Isto abre uma lacuna na área de tolerância a falhas, que motivou a realização deste trabalho.

1.2 Objetivo

O objetivo do trabalho é o projeto e a implementação de um serviço de detecção de falhas com semântica perfeita para redes locais, que ofereça garantias de qualidade de serviço. Este serviço, denominado Delphus[2], será implementado como um serviço a mais do sistema operacional, como o de autenticação e o de nomes. Também se propõe a exigir o mínimo de restrições do sistema, tornando sua adoção fácil e de baixo custo, em uma rede de pouca dispersão geográfica. Convém ressaltar que o projeto Delphus consiste não apenas desta dissertação, mas também do trabalho de Brito [13], cujo objetivo é desenvolver um módulo em *hardware* que ofereça suporte aos requisitos de sincronismo do serviço de detecção. Enquanto esta infra-estrutura não for concluída, o serviço utilizará um canal Ethernet privado para troca de mensagens.

1.3 Contribuições do Trabalho

O Delphus trará importantes contribuições para a área de sistemas distribuídos.

- Será um dos poucos detectores de falhas com semântica perfeita implementados. Além disso, diferentemente de outras implementações, sua abordagem impõe poucas restrições ao sistema e oferece garantias de qualidade de serviço.

- Permitirá que várias aplicações, que necessitam de serviços com esta semântica possam ser implementadas, com alto grau de confiabilidade.

[2]Na Grécia antiga localizava-se do oráculo de Delphus. Este era o mais confiável oráculo do seu tempo.

- Permitirá que algoritmos de consenso capazes de tolerar mais falhas possam ser adotados em aplicações distribuídas.

- Será um serviço de sistema operacional inédito, que oferecerá mais uma ferramenta de apoio para a implementação de técnicas de tolerância a falhas em sistemas distribuídos.

1.4 Organização do Trabalho

O restante do trabalho está organizado da seguinte forma. O Capítulo 2 apresenta os principais conceitos sobre Detecção de Falhas em Sistemas Distribuídos, imprescindíveis para a compreensão das seções seguintes. São apresentados os principais modelos de sistemas e de falhas, bem como a teoria de detectores de falhas não confiáveis. Neste capítulo, alguns outros exemplos de detectores de falhas são também descritos.

O Capítulo 3 descreve os modelos de sistema e de falhas adotados pelo serviço, seu funcionamento geral, arquitetura e o projeto da API. No Capítulo 4, são apresentados aspectos de implementação do serviço, tanto de *software* quanto de *hardware*. Os detalhes de como o módulo de sistema operacional, os canais de comunicação e as APIs do serviço foram implementados são vistos neste capítulo.

O Capítulo 5 apresenta um exemplo popular de utilização do Delphus: a replicação de servidores para tornar um serviço web tolerante a falhas.

O Capítulo 6 apresenta trabalhos de análise de desempenho do serviço. O impacto causado pelo Delphus no desempenho do sistema é analisado a partir dos resultados de três experimentos: teste do aumento do uso de memória RAM em função do aumento da carga do serviço, percentual de ocupação da CPU em vários instantes de tempo e um teste *Benchmark* do desempenho geral do sistema antes e depois da adoção do serviço.

Finalmente, conclusões e trabalhos futuros são apresentados no Capítulo 7.

2 Detecção de Falhas em Sistemas Distribuídos

A grande quantidade de atributos e regras que definem os sistemas distribuídos os torna particularmente complexos. Somente abstraindo tais complexidades, através de visões simplificadas, mas verossímeis destes sistemas, é que seu estudo se torna viável. Modelos são exemplos destas visões simplificadas. Através deles é possível estudar sistemas complexos, concentrando-se apenas em suas características principais.

Além disso, os sistemas distribuídos diferenciam-se bastante entre si, com relação a seus requisitos funcionais, carga a que são submetidos, requisitos de confiabilidade, características do ambiente de execução, quantidade de usuários, tipos de falhas a que estão sujeitos, dentre outros aspectos. O estudo destes sistemas requer a sua classificação em categorias, que agrupem exemplares com características semelhantes. Os modelos servem como classes, que permitem a abstração das diferenças menos relevantes, tornando mais fácil sua compreensão e a obtenção de soluções para problemas em comum.

Os sistemas distribuídos e os tipos de falhas a que estão expostos são divididos em vários modelos. A seguir, serão apresentados os principais modelos de falhas e de sistemas existentes.

2.1 Modelos de Falhas

Existem cinco modelos principais de falhas: o *crash* (por parada), o por omissão, o por valor, o temporal e o bizantino [26].

O modelo *crash* assume que um componente falha quando interrompe definitivamente seu funcionamento em um ponto específico de tempo e não realiza qualquer atividade depois disso. Se for reiniciado, não será capaz de recuperar um estado consistente anterior. Uma variação deste modelo é *crash-recovery*, que considera que o componente que falha por parada pode ser reiniciado e recuperar um estado consistente anterior à falha.

No modelo de falhas por omissão, um componente pára de produzir as respostas esperadas dele, mesmo que momentaneamente. Um sensor que ocasionalmente pára de produzir sinais, um canal de comunicação que perde mensagens e um processo operacional que não responde a mensagens dos demais são exemplos de componentes que comentem falhas por omissão.

No modelo de falhas por valor, um componente responde a uma entrada de valor, dentro de um intervalo de tempo correto, mas com um valor incorreto. Um processador que produz valores incorretos para cálculos, dentro do tempo esperado é um exemplo de um componente que falha por valor.

Existe também o modelo temporal, onde os componentes não atendem a requisitos temporais. Neste modelo, os componentes devem executar tarefas respeitando prazos ou intervalos de tempo. Quando não atendem a esses requisitos, são considerados falhos.

Por fim, no modelo bizantino, os componentes falham comportando-se de forma arbitrária, diferentemente de sua especificação, de forma não prevista pelos modelos anteriores. Por exemplo, um componente que produz uma resposta inesperada, sem ter havido uma entrada válida de dados, comete uma falha bizantina.

2.2 Modelos de Sistemas

Existem dois modelos principais de sistemas distribuídos: síncrono e puramente assíncrono. No entanto, várias pesquisas têm levado à definição de outros modelos com características híbridas entre estes dois, como por exemplo o modelo assíncrono equipado com um detector de falhas não confiável, que será explicado na Seção 2.3.

A seguir, estes dois modelos são apresentados. O critério usado nesta classificação teve como base os aspectos temporais do ambiente dos sistemas, que muito influenciam as técnicas de tolerância a falhas.

2.2.1 Modelo Síncrono

Os sistemas síncronos são aqueles em que é possível fazer considerações sobre atrasos máximos para entrega de mensagens na rede e sobre a velocidade da execução de processos [37]. Nestes sistemas, são controladas e limitadas todas as variáveis do ambiente, como número máximo de processos em execução, taxa de utilização da rede, velocidade de execução dos processadores e número de máquinas. Também supõe-se a existência de relógios sincronizados e com taxa de desvio limitadas e conhecidas. Este tipo de sistema se destina a aplicações críticas, com requisitos temporais e a redes que permitam o controle da carga do ambiente.

As aplicações projetadas para serem executadas em sistemas síncronos, baseiam-se em limites de comunicação específicos de cada sistema, o que diminui sua portabilidade.

Neste modelo, detectar a ocorrência de falhas do tipo *crash* é possível com a simples utilização de *timeouts* (tempos limites de comunicação). Neste tipo de sistema, é seguro afirmar que, se um processo não se comunica até um limite de tempo conhecido, ele terá falhado.

2.2.2 Modelo Puramente Assíncrono

O modelo puramente assíncrono não supõe a existência de quaisquer limites de atrasos para entrega de mensagens na rede, ou sobre qualquer outra variável do ambiente, como número de processos em execução ou desvio de relógios. A abrangência deste tipo de modelo é grande. A maioria dos sistemas práticos, incluindo a Internet, pode ser classificada como sistemas assíncronos. As soluções propostas para ele são aplicáveis aos demais modelos, uma vez que sua especificação não incorpora muitas restrições.

O resultado apresentado por Fischer *et al.* [30] prova ser impossível criar um protocolo que resolva, de forma determinista, o problema do consenso em um sistema puramente assíncrono sujeito a falhas. Isso se deve ao fato de que as incertezas no tempo de envio e entrega das mensagens tornam impossível uma conclusão precisa sobre o estado de um processo que não responde. Assim como ele pode ter falhado, ele pode apenas estar muito lento, ou a rede estar sobrecarregada e estar atrasando o envio das mensagens. Desta forma, o uso de *timeouts* não é uma solução precisa para detecção de falhas em sistemas que seguem este modelo. Esta dificuldade tem motivado a criação de novos modelos, como o de sistemas parcialmente–síncrono [25, 22] e assíncrono temporizado [21], que tentam contornar tal impossibilidade. Um dos novos modelos que ultimamente tem recebido mais atenção na literatura, é o do sistema assíncrono equipado com um detector de falhas não confiável.

2.3 Detectores de Falhas Nao Confiáveis

Inicialmente apresentados por Chandra e Toueg [16], os detectores de falhas não confiáveis são componentes que servem de oráculos ao sistema, informando os processos falhos. Tais

componentes fornecem uma abstração do mecanismo de detecção de falhas, que permite a simplificação dos algoritmos distribuídos, resumindo a detecção a simples consultas ao detector. O tipo de falha considerada é a do tipo *crash*, em que o processo interrompe definitivamente sua execução. Os detectores são geralmente organizados em módulos distribuídos pelo sistema, que monitoram o estado de subgrupos de processos e mantêm uma lista dos processos suspeitos de terem falhado. Cada processo tem acesso a um módulo local, através do qual pode consultar o estado dos demais processos. Os detectores de falhas são chamados de não confiáveis porque podem cometer erros e eventualmente adicionar nesta lista processos corretos. Se depois de um tempo, o módulo do detector descobrir que errou ao suspeitar de um determinado processo, este pode ser removido da lista de suspeitos.

O trabalho de Chandra e Toueg define e classifica os detectores de falhas através de propriedades abstratas ao invés de fornecer implementações específicas. Aspectos de implementação não são seu objetivo. Isso permite que as aplicações distribuídas sejam projetadas e sua correção provada, baseado apenas nessas propriedades, sem se comprometer com características de rede ou de software envolvidas na implementação.

As propriedades usadas para classificar os detectores são duas: abrangência e exatidão. Abrangência diz respeito a quantos módulos corretos conseguirão detectar que um processo falhou. Existem dois níveis de abrangência: forte e fraca. A abrangência fraca requer que após um tempo, a falha em um processo seja detectada por no mínimo, um módulo de detecção correto. A abrangência forte requer que após um tempo, a falha seja detectada por todos os módulos de detecção corretos. Exatidão diz respeito aos erros que um detector pode cometer. Existem quatro níveis de exatidão: forte, fraca, forte após um tempo e fraca após um tempo. A exatidão forte requer que os processos não sejam suspeitados por nenhum módulo de detecção antes que falhem. A exatidão fraca requer que pelo menos um processo não seja suspeitado antes que falhe. A exatidão forte após um tempo requer que a partir de algum tempo, a exatidão forte seja atendida. A exatidão fraca após um tempo requer que a partir de algum tempo, a exatidão fraca seja atendida. Através da combinação dessas propriedades, são definidas oito classes de detectores de falhas, como visto na Tabela 1.

Um outro conceito apresentado por Chandra e Toueg [16] é o de reducibilidade entre classes de detectores. Para fins de análise, as oito classes podem ser reduzidas para apenas

	Exatidão			
Abrangência	**Forte**	**Fraca**	**Forte após um tempo**	**Fraca após um tempo**
Forte	P	S	$\diamond P$	$\diamond S$
Fraca	Q	W	$\diamond Q$	$\diamond W$

Tabela 1: Classes de Detectores de Falhas

quatro, se considerarmos que um algoritmo pode ser empregado para fazer os detectores de uma classe se comportarem como os de outra. Por exemplo, se tivermos um sistema S que requeira a presença de um detector da classe D', mas só tivermos um detector da classe D, poderíamos utilizar um algoritmo A que fizesse o detector D se comportar como um detector D', conforme ilustrado na Figura 1. Desta forma, emularíamos D' para atender o sistema S. Quando isso ocorre, é dito que D' é redutível para D. Se D também for redutível para D', os dois são considerados equivalentes.

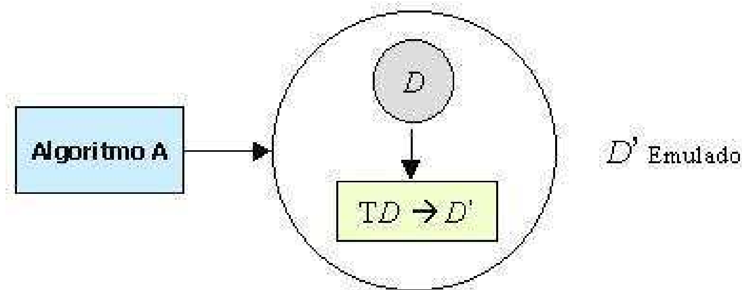


Figura 1: Reducibilidade de Classes de Detectores de Falhas [Fonte: Chandra e Toueg [16]]

As quatro classes de detectores de abrangência fraca (Q, W, $\diamond Q$ e $\diamond W$) são equivalentes às de abrangência forte (P, S, $\diamond P$ e $\diamond S$). Uma das reduções dispensa o emprego de algoritmos. Os de abrangência forte podem ser empregados em ambientes que esperam os de abrangência fraca, pelo motivo óbvio de que fornecem um serviço mais completo que os de abrangência fraca, podendo emulá-los sem dificuldade.

Os de abrangência fraca também podem emular os de abrangência forte. Neste caso, todos os seus módulos, e não apenas um, devem detectar que um processo falhou. Para isso, basta empregar um algoritmo que faça cada módulo enviar para os demais, a lista dos seus processos suspeitos. Cada módulo, ao receber as mensagens dos demais, adiciona a sua lista de suspeitos os suspeitos dos demais. Essa atividade é repetida indefinidamente. Se

alguma mensagem for perdida, outras em seguida serão tentadas, até que todos os módulos suspeitem de um processo falho. Se uma suspeita for errônea, o processo sairá das listas de suspeitos de todos os módulos e não será mais propagado nas mensagens. Desta forma, se um módulo detectar a falha em um processo, conseqüentemente, todos os demais o farão, em algum momento, exatamente o que caracteriza a abrangência forte. Este algoritmo permite que os detectores de abrangência fraca possam ser empregados em ambientes que esperam os de abrangência forte. Sendo assim, P, S, $\diamond P$ e $\diamond S$ são dedutíveis para os Q, W, $\diamond Q$ e $\diamond W$, respectivamente. Por causa desta equivalência, a literatura normalmente referencia apenas os detectores P, S, $\diamond P$ e $\diamond S$.

O que permite que classes com semânticas variadas possam ser implementadas, é o fato de que os sistemas distribuídos assíncronos práticos não são puramente assíncronos, possuindo algum nível de sincronia. Por exemplo, alguns sistemas utilizam redes locais que foram projetadas para suportar uma carga muito acima do que lhes é exigido de fato. Estes sistemas possuem ainda uma quantidade média de processos em execução e usuários ativos relativamente constante. Esses limites de utilização do sistema, mesmo que não sejam rígidos, acabam aumentando a previsibilidade dos tempos de comunicação entre os processos. Quanto maior for esta previsibilidade, maior será o grau de sincronia apresentado pelo sistema, e consequentemente, mais fortes serão as semânticas das classes possíveis de serem implementadas neste sistema. A Figura 2 ilustra a relação entre sincronismo e força da semântica dos detectores de falhas possíveis de serem implementados. Observe que o nível necessário para implementar um $\diamond S$ é bem menor do que para implementar um P.

Figura 2: Níveis de Sincronia em Sistemas Assíncronos

A classe $\diamond S$ atende as propriedades abrangência forte e exatidão forte após um tempo. O nível de sincronia necessário no sistema para que esta classe seja implementada é o mais

baixo de todos, possível de ser encontrado na maioria dos sistemas assíncronos práticos. Esta classe é a de semântica mais fraca que permite a solução de consenso de forma determinista em ambientes assíncronos, conforme demonstrado por Chandra *et al.* [17]. Este resultado e o fato de que esta classe não exige restrições fortes do ambiente para ser implementada tendenciaram a implementação de detectores de falhas com semânticas mais fracas, especialmente a $\diamond S$.

A classe P, que atende as propriedades abrangência forte e exatidão forte, é a que possui a semântica mais forte de todas. Os detectores desta classe devem ser capazes de suspeitar apenas dos processos que realmente falharem e ter um processo falho suspeitado por todos os seus módulos, dentro de algum tempo. Entretanto, detectores de falhas com semântica perfeita são implementáveis apenas em sistemas síncronos. Por causa desta limitação, o meio adotado para se oferecer um serviço de detecção de falhas perfeito a aplicações executando em ambientes assíncronos, é a adoção de restrições ao ambiente. Estas restrições devem criar um subsistema síncrono, que permita a garantia das propriedades da classe P. Elas se traduzem por alterações no sistema operacional, adoção de um canal extra de comunicação dedicado à troca de mensagens do serviço, ou ainda, controle do número de processos e máquinas suportados no sistema [45].

Os detectores de falha com semântica perfeita são os únicos que podem informar corretamente se um processo falhou ou não. Algumas aplicações, como sistemas redundantes primário–cópia, necessitam de detectores de falhas que não cometam erros. Quando o servidor cópia detecta a falha no servidor primário, ele deve assumir seu lugar, sem o risco do primário ainda estar operacional. Se um detector de semântica mais fraca fosse utilizado, uma suspeita incorreta poderia levar os dois servidores a se considerassem primários, podendo gerar conflitos.

2.4 Aspectos de Qualidade de Serviço

As classes dos detectores de falhas, mesmo a dos perfeitos, especificam comportamentos que devem ser observados após um tempo, pelos detectores de falhas. Por exemplo, após um tempo, todo processo que falha é permanentemente suspeito por um detector perfeito. Este

período de tempo não é quantificado pela especificação das classes dos detectores de falhas. Isso se deve ao fato de que tais especificações são destinadas a sistemas assíncronos, onde considerações sobre limites de tempo não são possíveis. Entretanto, mesmo em sistemas assíncronos, muitas aplicações possuem restrições temporais, mesmo que não mandatórias. Para estas aplicações, detectores de falhas que apresentam comportamento eventual não são suficientes. Eles permitem a solução de consenso de forma determinista, mas se não apresentarem um nível mínimo de qualidade de serviço, não permitirão que as aplicações respeitem suas restrições temporais [18].

Em se tratando de detectores de falhas não confiáveis, por qualidade de serviço entende-se uma especificação que quantifica a rapidez com que uma falha é detectada e o número máximo de falsas suspeições podem ocorrer [18]. Estas métricas servem como um complemento para a especificação de um detector de falhas, e também para se avaliar a sua usabilidade por aplicações com restrições temporais.

Em alguns casos, estas métricas podem tornar-se tão relevantes quanto as classes definidas por Chandra e Toueg. Por exemplo, saber que falhas são detectadas em no máximo um determinado período de tempo pode ser muito mais relevante em um sistema do que a garantia de que falsas suspeitas não ocorrem.

2.5 Implementações de Detectores de Falhas

Apesar de não poderem ser usados como uma indicação precisa da falha de um processo em um ambiente assíncrono, os *timeouts* são amplamente utilizados nas implementações de detectores de falhas. Isso se deve à simplicidade de seu uso e ao fato de que, em situações práticas, não é necessário esperar indefinidamente pela mensagem de um processo para considerá-lo falho. Basta para isso, esperar por um tempo suficientemente longo, em que a maioria dos processos operacionais consegue se comunicar no sistema. Isso possibilita inferir o estado do processo, com alta probabilidade de acerto.

Existem dois modelos baseados nestes limites de tempo para troca de mensagens: o *pull*, também chamado de *interrogation* e o *push*, também chamado de *heartbeat* [41]. O *pull* se baseia no envio periódico de mensagens do tipo *"você está vivo?"* para os processos

monitorados. Logo após o envio dessas mensagens, é ativado um temporizador e se o detector não receber de volta uma mensagem do tipo *"sim, eu estou vivo!"* daquele processo, este é adicionado na lista de processos falhos.

Já o método *push* baseia-se apenas no recebimento de mensagens do tipo *"eu estou vivo!"* dos processos, para considerar seu estado. Essas mensagens são comumente conhecidas como *heartbeats*. O não recebimento de tais mensagens, em um intervalo de tempo, faz com que o processo seja incluído na lista de suspeitos. Uma variação deste método é o *push ad hoc*. Nele, as mensagens do tipo *"eu estou vivo!"* são enviadas apenas quando a detecção da falha é relevante. Por exemplo, um processo p pode requisitar um serviço de q, invocando uma função f. Neste momento, q começa a enviar mensagens do tipo *"eu estou vivo!"*, para que o detector de falhas o diagnostique como correto, no caso de p consultar o estado de q enquanto espera pela finalização de f. Em quaisquer dos métodos, se uma mensagem de um processo chega após ele ser considerado suspeito, então ele é retirado da lista de suspeitos.

Vale ressaltar que existe ainda uma quarta abordagem para a detecção de falhas em aplicações distribuídas: a implementação *ad hoc* [41], sem mensagens específicas para a detecção de falhas. Na verdade, ela não utiliza detectores de falhas para consultar o estado dos processos e a aplicação implementa seu próprio modo de detecção de falhas. Ela baseia-se no fato de que um algoritmo precisa da informação de detecção de falhas apenas em alguns momentos muito específicos. Nesses momentos, o algoritmo envia uma mensagem para o processo do qual a informação de falha é importante. Essa mensagem é chamada mensagem crítica e é uma mensagem normal, com significado para os algoritmos da aplicação. O processo que enviou a mensagem crítica aguarda então uma resposta crítica. Se não receber esta resposta crítica até um determinado momento, o processo que enviou a mensagem crítica marca o processo destino desta mensagem como suspeito. Esta abordagem exige que o algoritmo conheça, de antemão, as mensagens que serão enviadas e tire proveito do grau de sincronismo do canal. Assim como os detectores de falhas não confiáveis, esta técnica não pode ser implementada em sistemas assíncronos puros.

Desde o trabalho de Chandra e Toueg [16], vários detectores de falhas têm sido implementados [39, 19, 24, 7, 8, 36, 29]. Alguns deles seguem completamente os modelos e especificações apresentadas, outros tentam abordar aspectos não contemplados por ele. Sampaio *et*

al. descrevem várias implementações de detectores de falhas em um tutorial sobre detectores de falhas em sistemas assíncronos [40]. A seguir, veremos uma breve descrição de algumas implementações de detectores de falhas.

2.5.1 ESPRIT OpenDREAMS

Felber *et al.* [27] propõem que o detector de falhas seja um serviço do sistema operacional, assim como os serviços de nomes, de autenticação e de gerenciamento de arquivos. O detector de falhas seria inacessível diretamente e seu serviço estaria acessível a desenvolvedores através de objetos e um grupo de interfaces, organizadas hierarquicamente, apresentadas na Figura 3.

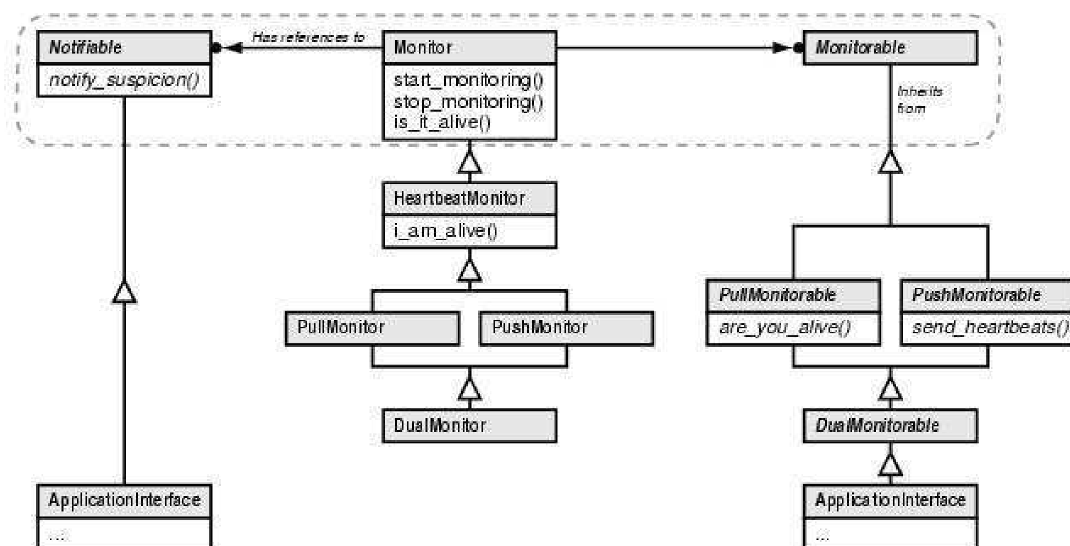


Figura 3: Arquitetura do ESPRIT OpenDREAMS [Fonte: Felber *et al.* [27]]

Ao invés de considerar a monitoria de processos, Felber *et al.* [27] considera a monitoria e a notificação de objetos. Isso não impede, entretanto, que um processo seja monitorado ou notificado. Basta que um objeto haja como seu representante e intermedeie a comunicação do processo com o detector de falhas.

Os papéis no processo de monitoramento são definidos como: *Monitor*, *Monitorable* e *Notifiable*. *Monitor* é o objeto central que representa o detector de falhas e que oferece uma interface para o uso do serviço. *Monitorable* é todo objeto que deva ser monitorado para

detecção de falhas e *Notifiable* é todo objeto que deva ser notificado da ocorrência de falhas nos objetos monitorados.

O objeto *Monitor* fornece uma interface para que clientes possam iniciar ou interromper o serviço de monitoria, consultar o estado de objetos monitoráveis gerenciar as listas de objetos notificáveis e monitoráveis e realizar a notificação dos objetos notificáveis. Para tornar-se um objeto notificável, um objeto deve implementar a interface *Notifiable* e cadastrar-se na lista de notificáveis do objeto *Monitor*. Para tornar-se um objeto monitorável, um objeto deve implementar uma sub interface da interface *Monitorable*: *PullMonitorable*, *PushMonitorable* ou *DualMonitorable*. Este modelo permite que tanto a abordagem do modelo *pull* quanto do modelo *push*, ou mesmo as duas sejam adotadas pelo detector de falhas, a depender das características do sistema e de cada objeto monitorado.

Mara monitorar um objeto, O *Monitor* adotará o modelo *pull*, caso a interface implementada seja *PullMonitorable* ou *push*, caso a interface implementada seja *PushMonitorable*.

Este trabalho propõe também um segundo modelo, que mescla as duas abordagens dos dois modelos tradicionais *push* e *pull*. Ele é adotado para os objetos que implementam a interface *DualMonitorable*, e é composto por duas fases. Na primeira, o objeto *Monitor* aguarda o envio das mensagens *"eu estou vivo!"*, seguindo o estilo do modelo *push*, vindas do objeto monitorado. Passado um limite de tempo, se nenhuma mensagem for recebida, então na segunda fase, o *Monitor* envia uma mensagem *"você está vivo?"* e espera uma mensagem *"sim, eu estou!"*, características do modelo *pull*. Apenas se esta segunda mensagem também não for recebida, o objeto será considerado falho. Caso seja recebida a mensagem, novamente o Monitor executará a primeira fase, e assim por diante. Ele permite que o modelo *pull* seja adotado apenas quando o *push* falhar. Desta forma, este protocolo permite que menos mensagens sejam trocadas na rede, em comparação com o modelo *pull*. Entretanto, o tempo de latência da detecção da falha se torna maior, atrasando a detecção de falhas pelo serviço.

Este modelo foi implementado como o serviço de detecção de falhas, parte do projeto *ESPRIT OpenDREAMS*, desenvolvido pelo laboratório de sistemas distribuídos do Swiss Federal Institute of Technology, na Suíça. O *ESPRIT OpenDREAMS* é um *framework* compatível com CORBA, para supervisão e controle de sistemas distribuídos, que tem influenciado várias propostas feitas à OMG para padronização de mecanismos de tolerância a

falhas para o CORBA [3].

2.5.2 Monitor

O Monitor é uma ferramenta desenvolvida pelo Laboratório de Sistemas Distribuídos da Universidade Federal da Bahia, utilizando a plataforma Java/CORBA. Ele é composto por dois serviços: o SDF - Serviço de Diagnóstico de Falhas [10] e o SGD - Serviço de Gerenciamento Distribuído [23].

O SDF é uma implementação de um detector de falhas da classe $\diamond S$ que adota o modelo de detecção *pull*. Ele permite que outras aplicações se cadastrem como interessados em receber notificações da ocorrência de eventos envolvendo os processos monitorados. Alguns exemplos de notificação são: processo falho, inclusão de um novo processo para monitoria, exclusão de um processo da monitoria, falha em uma máquina e finalização normal de um processo. O SGD é um dos interessados em receber estas notificações. Ele tem a função de permitir o gerenciamento dos processos de uma aplicação distribuída, permitindo a um administrador criá-los em qualquer máquina da rede, terminar os que estiverem falhos, movê-los de uma máquina para outra, configurar suas propriedades, além de fornecer uma visão consistente de todas as máquinas e processos monitorados. O SGD pode ser iniciado em qualquer estação da rede, desde que possa consultar o serviço de nomes CORBA e conseqüentemente um módulo SDF. Esta integração dos módulos do Monitor é ilustrada na Figura 4. Ela mostra uma rede onde três máquinas executam módulos do SDF. Esses módulos monitoram remotamente os processos P1, P2, P3, P4, P5 e P6, em execução nas máquinas. Uma outra máquina executa um módulo do SGD e recebe dos módulos SDF a informação de ocorrência de eventos com os processos monitorados. Usando este modulo do SGD, um administrador pode enviar comandos de gerenciamento para as outras máquinas, como a criação de um processo P7, e a finalização do processo P1. Uma quinta máquina é um servidor de nomes CORBA, que fornece as referências remotas dos objetos envolvidos nos dois serviços, que permitem seu funcionamento.

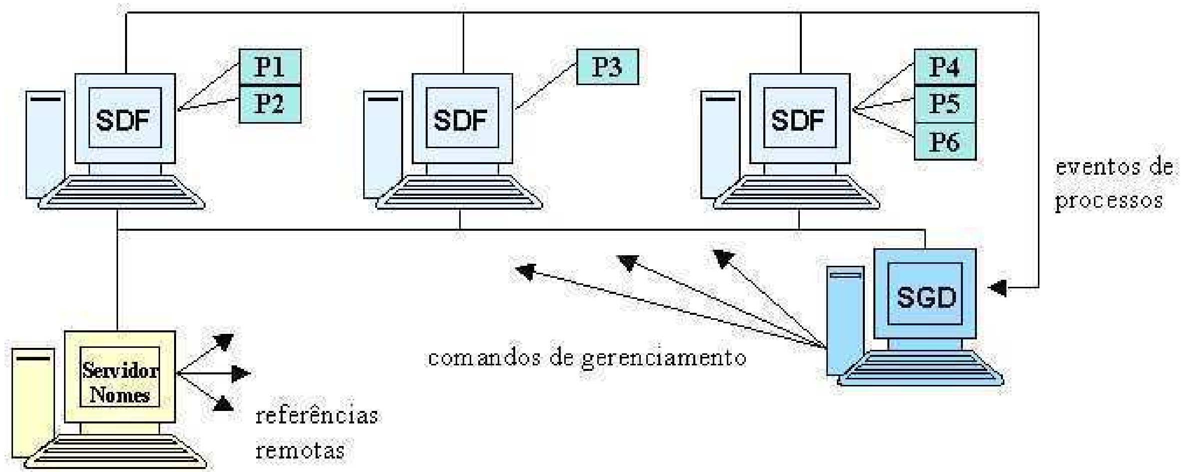


Figura 4: Monitor em Funcionamento

O SDF continua funcionando mesmo que a maioria de seus módulos falhe e reste apenas um ativo. Assim que é iniciado, um módulo procura por outro módulo SDF presente na rede. Uma vez encontrado, requisita dele a lista de todos os processos monitorados. Obtendo esta lista, passa então a monitorar todos os processos, de forma autônoma. Por isso, a falha de um módulo não implica na interrupção da monitoria de nenhum processo, mesmo daqueles em execução na máquina do módulo que falhou.

Um módulo do SDF é dividido em dois sub-módulos que trabalham de forma coordenada: o módulo de Reconfiguração e o módulo de Detecção de Falhas. O módulo de Reconfiguração funciona como o cérebro do SDF. Ele é responsável por manter a lista de processos corretos atualizada, iniciar a monitoria de processos, interromper a monitoria de processos detectados como falhos, e notificar a ocorrência de eventos de processos a outros objetos que se cadastrem como interessados nesta notificação.

O módulo de detecção de falhas, como o próprio nome diz, tem a função de detectar a ocorrência de falhas em processos ou máquinas onde os processos residem. Ele é responsável por manter informações sobre os níveis de tempo de comunicação aceitáveis para os processos monitorados, que são definidos durante o cadastro de cada processo. Além disso, tem a função de tentar periodicamente contactar cada processo monitorado e trocar mensagens com as máquinas onde residem, a fim de detectar processos que não estejam respondendo

dentro dos limites de tempo aceitáveis, seja por falha na máquina ou neles mesmos.

Cada módulo SDF possui suas próprias listas de processos corretos e de processos falhos e elas tendem a possuir o mesmo conteúdo. Em algumas situações, podem não possuir a mesma informação, mas protocolos *ad hoc* garantem a convergência de todas as listas para uma visão única e coerente do estado dos processos.

Existem três formas para que um processo seja monitorado pelo SDF. Se for feito em Java, ele pode implementar uma interface de gerenciamento definida pelo serviço. Para averiguar se o processo está em funcionamento, o SDF invoca periodicamente o método *respondeMonitoria* definido nesta interface. Caso seja desenvolvido em outra linguagem, é necessário o uso de um processo intermediário que implemente a interface de gerenciamento do SDF e que se comunique com o processo monitorado. Estes processos devem portanto, possuir um processo intermediário para que possa ser monitorado pelo SDF. A terceira possibilidade é a de que a monitoria seja feita sobre um objeto que siga a especificação CORBA Tolerante a Falhas [3]. Neste caso, também é necessário o uso de um processo intermediário que implemente a interface de gerenciamento do SDF e que conheça a interface de gerenciamento prevista pelo FT CORBA, para se comunicar com o objeto monitorado.

2.5.3 Licenças e Watchdogs

O trabalho de Fetzer [28] propõe um protocolo baseado no uso de *watchdogs* em hardware e de licenças de funcionamento para construir um detector de falhas perfeito. Este é um protocolo simétrico, ou seja, todas as máquinas participantes executam os mesmos procedimentos e considera a existência de apenas três máquinas participantes.

Cada máquina deve possuir acoplado um *watchdog*, um componente em *hardware*, programado para desligá-la em um instante específico de tempo. Antes que chegue esse instante, a máquina deve obter, de uma das duas outras máquinas, uma licença que a autorize continuar funcionando. Ao receber esta licença, a máquina adia o tempo limite do *watchdog* para não ser desligada. Quando essa licença não é obtida, o tempo limite se esgota e o *watchdog* desliga a máquina, forçando-a a falhar. As duas outras máquinas devem considerar a máquina em questão correta apenas se a última requisição de licença recebida dela não tiver uma "idade"

superior ao tempo limite dos *watchdogs*. Esta idade é calculada através da utilização do serviço *fail aware datagram*, que utiliza o mínimo de sincronismo do sistema para determinar se uma mensagem foi entregue dentro de um determinado limite de tempo [29]. Isso permite calcular o momento em que uma mensagem foi enviada e consequentemente, sua idade aproximada. Além do mais, estas duas máquinas comunicam-se entre si para identificar qual das duas concedeu a última licença à terceira máquina e assim, calcular a idade correta.

Caso a máquina realmente falhe, ela não requisitará mais licenças e assim, as demais acabarão diagnosticando-a como falha. Este protocolo garante também que se, por um motivo qualquer, a máquina não conseguir renovar sua licença, ela terá realmente falhado no momento da expiração de sua última licença que lhe foi conhecida. Desta forma, as suspeições das demais máquinas estarão corretas.

Se fosse preciso utilizar este protocolo em mais de três máquinas, várias instâncias dele teriam que ser produzidas, agrupando as máquinas que precisassem detectar falhas entre si. Se uma máquina executasse mais de uma instância do protocolo, cada uma delas atualizaria o temporizador de um *watchdog* em *software*. Quando o temporizador do *watchdog* em *software* de uma instância expirasse, o *watchdog* em *hardware* da máquina seria acionado para desligá-la, provocando a detecção de sua falha por todas as demais máquinas.

Infelizmente este protocolo só considera a monitoria de máquinas e não considera aspectos de qualidade de serviço.

2.5.4 TCB (Timely Computing Base)

Um dos principais trabalhos relacionados à implementação de detectores de falhas perfeitos é o TCB, apresentado por Veríssimo e Casimiro [45, 15].

Na verdade, o TCB tem objetivos muito além do fornecimento de um serviço de detecção de falhas perfeito. Ele é um *framework*, que fornece suporte para a execução de aplicações com requisitos temporais em sistemas assíncronos. Estas aplicações podem requisitar que suas tarefas sejam executadas respeitando prazos para conclusão, ou em instantes exatos de tempo. Este suporte permite o acompanhamento da execução das aplicações e permite a detecção de falhas temporais, que englobam as falhas do tipo *crash*. Por isso, a implementação

de um detector de falhas perfeito torna-se possível e desta forma, a detecção de falhas com semântica perfeita também está entre os serviços disponibilizados pelo TCB.

Entretanto, para que todo este suporte seja fornecido, o TCB necessita que fortes restrições sejam adotadas no sistema. O sistema assumido é composto de uma parte assíncrona, onde aplicações sem garantias de prazo executam utilizando uma rede assíncrona, e uma porção síncrona, onde executa o TCB.

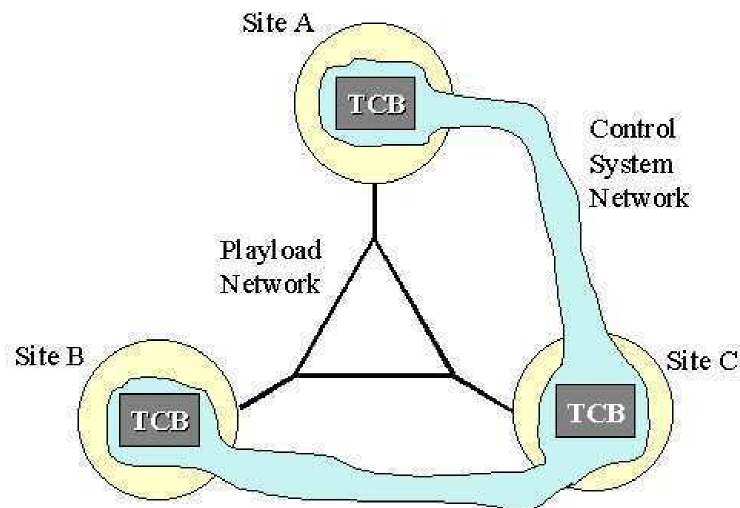


Figura 5: Arquitetura do TCB [Fonte: Veríssimo e Cassimiro [45]]

A base para o funcionamento do TCB é a existência de um módulo síncrono dentro de cada máquina, não importando quão síncrono seja o restante do sistema. Além disso, os módulos do TCB se comunicam por uma rede privada síncrona.

Uma implementação do TCB foi construída a partir de um módulo de software que executa no sistema operacional de tempo real Real Time Linux em cada máquina. Conforme visto na Figura 5, as máquinas utilizam uma rede síncrona privativa do TCB para comunicação, chamada *Control System Network*. As máquinas são também conectadas por um canal assíncrono, chamado *Payload Network*, onde ocorre a comunicação assíncrona entre as aplicações do sistema.

Antes de iniciar, as aplicações devem informar o prazo e o instante que suas tarefas deverão ser iniciadas, e ao terminá-las, informar que foram concluídas. Isto faz com que as

aplicações tenham que ser escritas especificamente para o uso do TCB o que aumenta seu grau de acoplamento com o *framework* e diminui sua portabilidade.

3 Projeto do Serviço Delphus

3.1 Modelos de Sistema e de Falhas

O sistema considerado neste trabalho é formado por um ambiente assíncrono, equipado com um subsistema síncrono [45]. O ambiente assíncrono é o ambiente padrão onde o sistema operacional e as aplicações executam e trocam mensagens. O canal de comunicação que serve a este ambiente está sujeito a particionamento da rede e perda de mensagens.

O subsistema síncrono é um ambiente especial, embutido no sistema assíncrono, que fornece o grau de sincronismo necessário para o funcionamento do serviço Delphus. É considerado que não ocorrem perda de mensagem, nem particionamento de rede no canal de comunicação usado por este subsistema.

A Figura 6 apresenta os dois sistemas. Nela é possível ver o sistema assíncrono formado pelo sistema operacional, processos e a rede local. Embutido neste ambiente, está o subsistema síncrono, utilizado pelos módulos de detecção de falhas, presentes em cada máquina da rede.

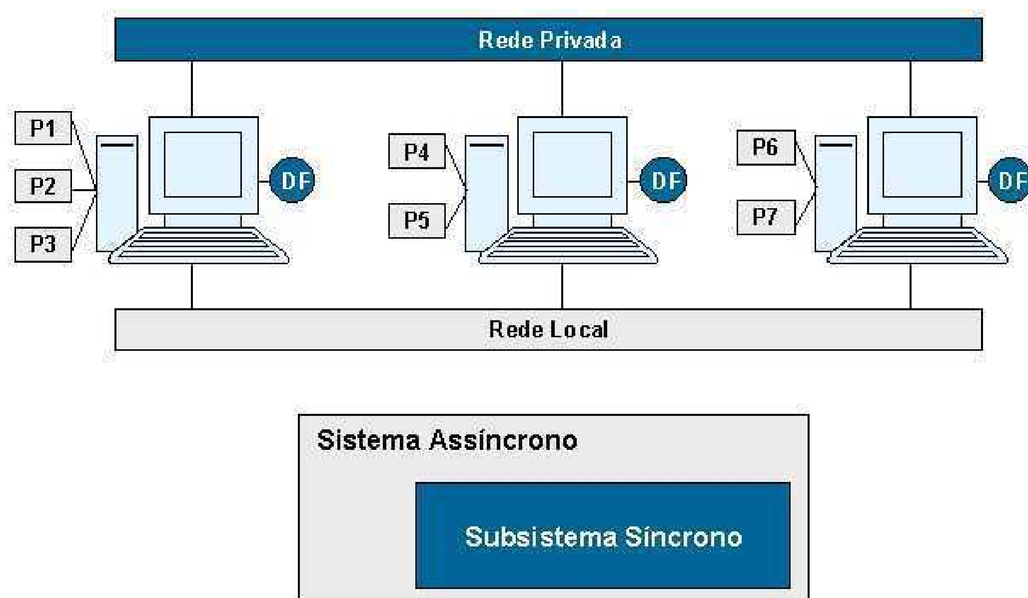


Figura 6: Modelo do Sistema

O conjunto de máquinas da rede que executam módulos do serviço (um por máquina) é

chamado de *domínio de detecção*. No momento, apenas um domínio de detecção é considerado, mas trabalhos futuros poderão ampliar a atual arquitetura de domínio único para uma arquitetura multi-domínio.

O modelo de falhas considerado pelo Delphus é o por parada (*crash*), onde uma entidade que falha, pára sua execução definitivamente e se for reiniciada, não terá memória de sua execução passada.

3.2 Arquitetura do Serviço

A arquitetura do serviço, vista na Figura 7, é composta por três camadas: interface, controle e comunicação.

Figura 7: Camadas da Arquitetura do Delphus

A camada de comunicação é responsável por permitir a troca de mensagens entre os módulos do serviço. Esta camada é formada pelo canal assíncrono da rede local, de uso genérico pelo sistema, e pelo canal síncrono, de uso privativo do serviço. A camada de controle é responsável por executar as funcionalidades essenciais do serviço. A camada de interface é a responsável por permitir o uso do serviço pelas aplicações clientes.

As entidades que usam o serviço são classificadas em dois tipos: Monitoráveis e Notificáveis, conforme apresentado por Felber *et al.* [27]. Monitoráveis são entidades (máquinas, processos, objetos, etc.) que devem ser monitoradas pelo serviço. Notificáveis são entidades (processos, objetos, etc.) que devem receber uma notificação quando determinado evento de monitoria (uma falha por parada, por exemplo) ocorrer em um Monitorável. Todas as entidades associadas a um domínio de detecção em particular são identificadas por um identificador global, exclusivo em sua categoria (monitoráveis ou notificáveis).

O serviço aborda aspectos de qualidade de serviço [18]. Considerando que o Delphus é um serviço de detecção de falhas com semântica perfeita, podemos dizer que o valor da métrica de quantas falsas suspeições o serviço realiza é zero. Com relação à métrica do período de latência de detecção de falhas, o serviço permite que se determine para cada entidade monitorável, o tempo limite em que as falhas devem ser detectadas, como será explicado mais adiante nesta seção e na Seção 4.

A Figura 8 apresenta os três elementos principais que compõem a camada de controle: o Escalonador, o Coração e o Notificador. O Escalonador permite que entidades sejam cadastradas para serem monitoradas pelo serviço. Ele também programa quando os *heartbeats* das entidades monitoradas deverão ser transmitidos, de forma a manter as garantias de qualidade de serviço e racionalizar o uso do canal síncrono. O Coração é responsável por averiguar o estado das entidades monitoráveis locais, gerar suas mensagens *heartbeats*, que informam seus estados, e transmití-las pela rede síncrona para os demais módulos do serviço. O Notificador tem três funções: ouvir as mensagens *heartbeat* na rede síncrona, permitir o cadastro de Notificáveis e notificar tais entidades sobre a ocorrência de eventos nos quais estão interessadas.

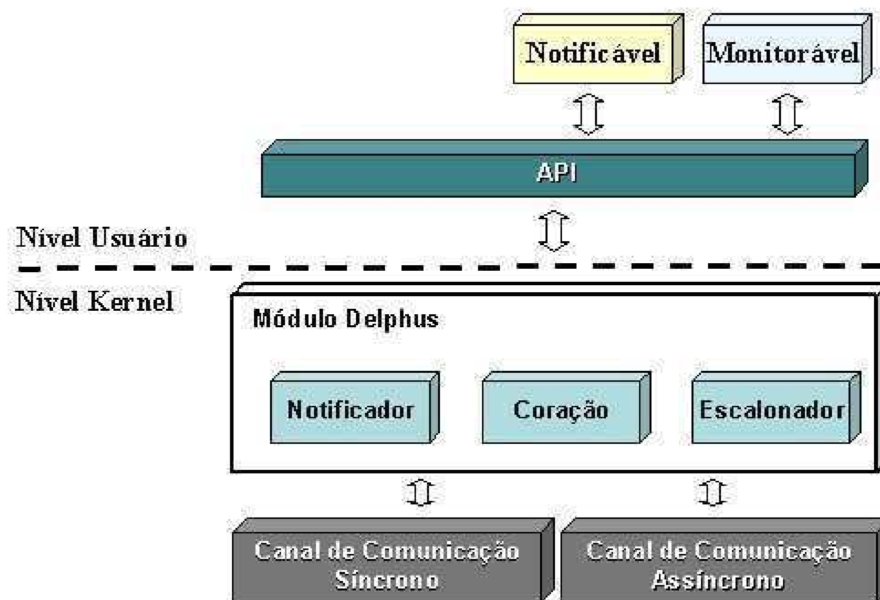


Figura 8: Arquitetura do Delphus

3.3 Descrição do Serviço

Cada módulo do Delphus implementa um detector de falhas do estilo *push* [27] que periodicamente envia mensagens de *heartbeat* de suas entidades monitoráveis locais. Essas mensagens são enviadas pela rede síncrona.

A utilização desta rede síncrona obedece um protocolo TDMA (*Time Division Multiple Access*) [6, 42]. No TDMA, a utilização do canal é dividida em fatias de tempo, também conhecidas como *slots*. Como visto na Figura 9, cada fatia de tempo é usada por apenas uma máquina. As máquinas seguem uma ordem de utilização do canal conhecida por todas. A sequência de utilização do canal se repete ciclicamente, e a estes ciclos dá-se o nome de períodos TDMA.

Cada módulo do Delphus tem sua própria fatia do período TDMA para a transmissão das mensagens de *heartbeat* das entidades monitoráveis locais. Sempre que um *heartbeat* de uma entidade monitorável está ausente, todo módulo correto detecta a falha desta entidade. Então, cada módulo notifica todas as entidades notificáveis locais que solicitaram uma notificação para a falha daquela entidade monitorável cuja falha foi detectada. Cada módulo tem ao menos uma entidade monitorável; esta entidade é a máquina na qual o módulo executa. Desta forma, em cada período, todo módulo deve utilizar sua fatia de tempo no canal para transmitir ao menos o *heartbeat* da sua máquina.

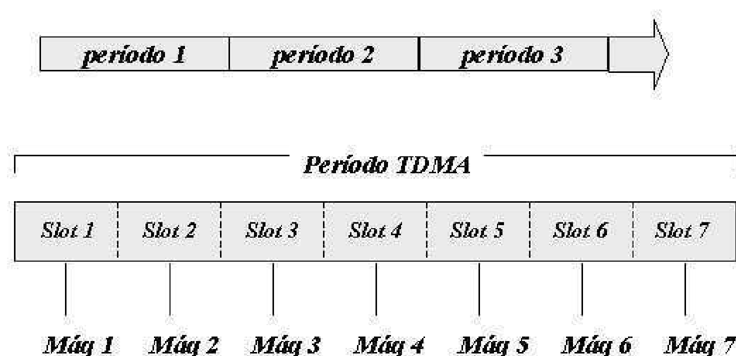


Figura 9: Protocolo de Comunicação TDMA

Os módulos independentes precisam estar sincronizados entre si para implementar o protocolo TDMA. Para isso, eles utilizam um protocolo baseado em um líder. Sempre que

houver ao menos um módulo executando o serviço, haverá exatamente um módulo como líder. Além de enviar os *heartbeats* de suas entidades monitoráveis locais, como qualquer máquina do domínio, o líder é responsável também pelo envio de uma mensagem de sincronização no início de cada período TDMA. Esta mensagem de sincronização é utilizada pelos outros módulos para a ressincronização.

Após a mensagem de sincronização, todos os membros do domínio devem escutar uma outra mensagem enviada pelo líder no canal síncrono: a mensagem de anúncio. Cada mensagem de anúncio contém informação sobre uma única inscrição ou remoção. O anúncio da inclusão de um novo monitorável contém toda informação relevante para definir a identificação da entidade e a frequência com que os *heartbeats* daquela entidade precisam ser transmitidos. Por outro lado, o anúncio de uma remoção contém apenas a identificação da entidade a ser removida. Depois de recebido o anúncio, todos os membros atualizam suas listas de monitoráveis, assegurando que todos têm o mesmo conjunto de monitoráveis, numa mesma ordem e a mesma informação de configuração para eles. Essa equivalência entre as listas é utilizada para definir em qual fatia de tempo do período TDMA o módulo deve transmitir seus *heartbeats*.

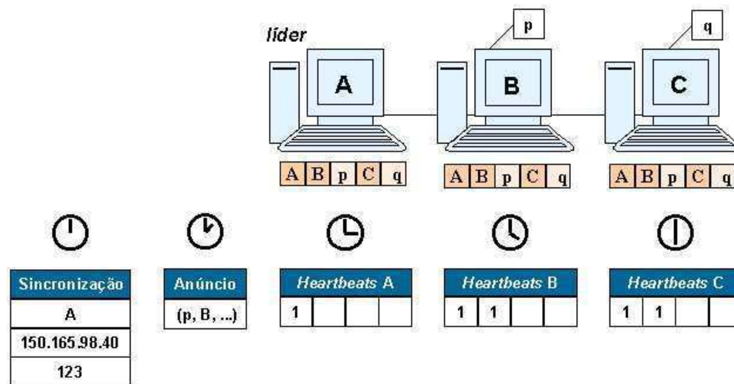


Figura 10: Protocolo de Comunicação TDMA do Delphus

No exemplo de um período TDMA do Delphus, exibido na Figura 10, o domínio de detecção é composto por três máquinas: A, B e C. A máquina A é a líder do domínio e por isso, envia as mensagens de sincronização e anúncio nas duas primeiras fatias TDMA. A mensagem de anúncio, neste exemplo, informa a admissão de um processo em execução

na máquina B como monitorável, identificado no domínio de detecção como p. Após o envio das mensagens do líder, cada uma das três máquinas usa uma fatia TDMA para transmitir mensagens de *heartbeat*. O módulo da máquina A transmite apenas um *heartbeat*, referente à máquina onde executa. Os módulos das máquinas B e C enviam dois *heartbeats* cada, referentes às suas máquinas e aos processos locais p e q, respectivamente. Como cada módulo do Delphus possui a mesma lista de monitoráveis, numa mesma ordem (A, B, p, C, q), eles têm como identificar o usuário de cada fatia TDMA e a que monitorável se refere cada um dos *heartbeats*.

3.3.1 Eleição de Líder

A utilização de um protocolo baseado em líder introduz dois problemas a serem resolvidos: i) como definir quem é o primeiro líder, na iniciação do serviço; e ii) como lidar com falhas do líder.

O principal problema enfrentado pelo serviço durante a fase de iniciação é descobrir se uma máquina é a primeira a se inscrever em um domínio de detecção ou não. Se esta máquina for a primeira, ela deve assumir a liderança do domínio e tornar-se responsável pela coordenação dos períodos TDMA. Caso contrário, ela deve encontrar o líder e requisitar sua inscrição no domínio. A solução adotada para solucionar este problema é dependente da tecnologia adotada para implementar o canal síncrono. Por este motivo, o protocolo utilizado para a iniciação do serviço não é descrito em detalhes aqui, mas na Seção 4.2.2.

Tolerar falhas do líder é uma tarefa mais simples. Falhas do líder impactam o protocolo apenas quando há ao menos um outro módulo correto em operação no mesmo domínio de detecção. Logo, a ausência de uma mensagem de sincronização pode ser utilizada para indicar a falha do líder. Como explicado anteriormente, após o estabelecimento do líder no procedimento de iniciação, todo novo módulo precisa explicitamente requisitar ao líder sua inscrição no domínio. O líder responde tal requerimento anunciando todo novo membro, um em cada período TDMA. Este anúncio utiliza uma fatia TDMA, e é recebido por todo membro que já está inscrito (além do membro que se inscreve). Desta forma, este procedimento estabelece uma ordenação para os módulos que já se inscreveram no serviço. Esta

ordenação é utilizada para definir o novo líder no momento que uma falha do líder corrente é detectada. A Figura 11 ilustra esta situação. Os blocos em cinza representam as mensagens não recebidas, por conta de uma falha na máquina A, líder do domínio.

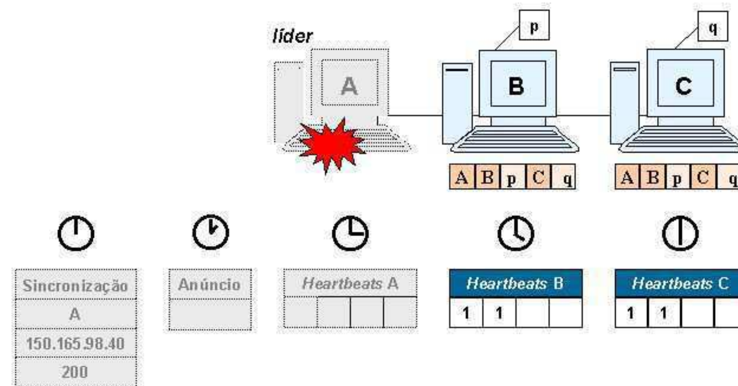


Figura 11: Falha do Líder do Domínio de Detecção

O módulo escolhido para ser o novo líder imediatamente se torna ciente sobre seu novo papel. Na próxima fatia TDMA de sincronização, ele irá começar a enviar mensagens de líder (sincronização e anúncio), como mostrado na Figura 12.

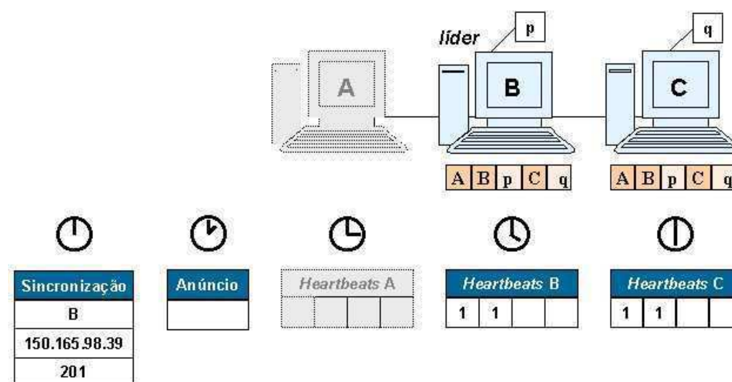


Figura 12: Eleição de Líder do Domínio de Detecção

3.3.2 Admissão de Monitoráveis

Antes de realizar os anúncios, o líder precisa receber requerimentos de inscrição ou remoção dos módulos. Para aliviar o tráfego na rede síncrona, a rede assíncrona é utilizada para

permitir toda a comunicação necessária antes da realização do anúncio. Um requerimento para inscrever uma máquina na lista de monitoráveis é realizado sempre que o serviço é iniciado na máquina, enquanto que um requerimento de remoção para uma máquina é feito sempre que o serviço é encerrado. Depois que o serviço é iniciado em uma máquina em particular, entidades naquela máquina podem solicitar inscrições ou remoções. Estes requerimentos são endereçados ao módulo local que os redireciona para o líder. Antes de redirecionar uma requisição de inscrição de uma entidade local, cada módulo executa um algoritmo de escalonamento (explicado em detalhes na Seção 3.3.3) que verifica se a qualidade de serviço requisitada (expressa pelo limite superior da latência de detecção requerida) pode ser garantida. Se não, a função de inscrição retorna um erro para o processo solicitante. Caso contrário, o módulo continua o procedimento de inscrição.

Papel do Líder

Quando um líder recebe um requerimento de inscrição através da rede assíncrona, ele verifica se a identificação requisitada já está em uso. Se estiver, uma mensagem de recusa é enviada para o requisitante e o procedimento de inscrição é encerrado. Caso contrário, a ação tomada irá depender do tipo de entidade que está requisitando a inscrição. Se a entidade monitorável não for uma máquina, o líder a escalona para ser anunciada na próxima mensagem de anúncio disponível. Caso contrário, o líder envia ao módulo requisitante uma lista com todas as entidades monitoráveis correntes. De posse desta lista, o módulo atualiza seus dados locais, envia uma mensagem de confirmação para o líder, e então espera pelo seu anúncio no próximo período.

Papel do Módulo

A partir do momento que um módulo envia sua primeira requisição (relativa a sua máquina), ele começa a escutar mensagens de anúncio. Isto garante que sua informação local, construída após o recebimento da lista de entidades monitoráveis do líder, será consistente com a lista de todos os outros módulos. Apenas após receber a mensagem de confirmação, o líder escalona o anúncio da entidade máquina. Em todos os casos, o procedimento de inscrição/remoção é concluído apenas quando o anúncio correspondente é recebido através

do canal síncrono, ou uma mensagem de recusa é recebida através do canal assíncrono.

Como um canal assíncrono é utilizado para enviar as mensagens do procedimento de inscrição/remoção, é possível que algumas delas sejam perdidas ou atrasadas. Dessa forma, o módulo de origem da requisição usa um temporizador para detectar quando deve retransmitir uma mensagem. Eles devem também estar preparados para descartar duplicatas. Além disso, no caso de particionamento da rede assíncrona, um número máximo de retransmissões pode ser alcançado, e o procedimento de inscrição/remoção irá terminar e retornar um erro para o módulo requisitante.

Exemplo de Admissão

A Figura 13 mostra um exemplo de admissão de monitorável. Uma máquina B tenta ingressar em um domínio de detecção, do qual apenas a máquina A faz parte como líder.

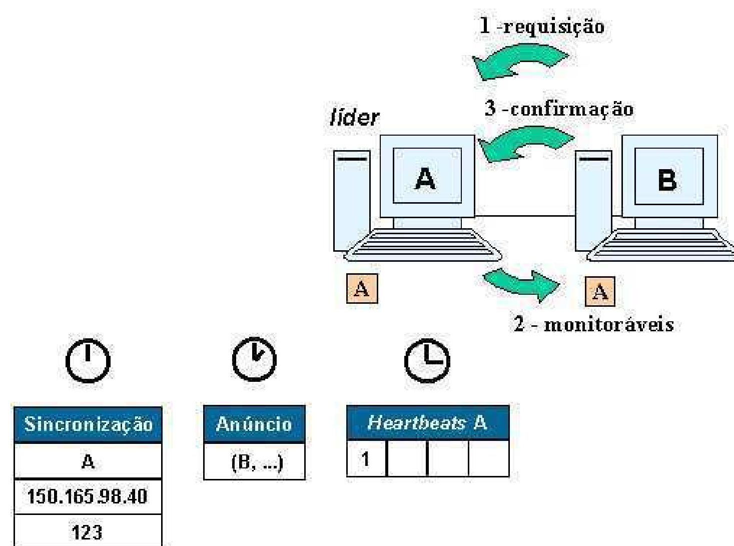


Figura 13: Admissão de uma nova máquina

O primeiro passo seguido por B é enviar uma mensagem pela rede assíncrona requisitando admissão. Nesta mensagem, B informa seu identificador pretendido e seu endereço IP na rede assíncrona. Recebendo esta mensagem, A consulta se o identificador pretendido por B é exclusivo. Caso afirmativo, A envia para B a lista de monitoráveis que atualmente fazem parte do domínio de detecção. Após receber esta lista, B envia para A uma mensagem de

confirmação. O último passo seguido por A é anunciar o ingresso de B na fatia de anúncio do próximo período TDMA. Somente após receber esta mensagem de anúncio, B se considera parte do domínio e começa a usar uma fatia para transmitir suas mensagens de *heartbeat*.

3.3.3 Escalonamento de Transmissão de Heartbeats

Como cada máquina utiliza uma única fatia TDMA para enviar as informações de *heartbeats* de todos os seus monitoráveis em cada período TDMA, e cada *heartbeat* requer um tempo para ser devidamente remetido e recebido, o número máximo de monitoráveis atendidos por uma única máquina é limitado.

Desta forma, uma fatia de tempo TDMA para *heartbeats* é dividida em um número fixo de sub-fatias. De modo a incrementar a disponibilidade de sub-fatias de tempo livres em cada período TDMA, o serviço escalona monitoráveis para enviarem seus *heartbeats* na menor frequência possível. Esta frequência deve ser alta o suficiente para ainda garantir que as suas falhas serão detectadas com o limite de latência desejado. Assim, nem todos os monitoráveis têm os seus *heartbeats* transmitidos todo período. Eles são escalonados para serem enviados apenas quando estritamente necessários para prover a latência de detecção necessária. Este procedimento também raciona o uso do canal de comunicação síncrono, diminuindo ainda mais a possibilidade de sobrecarga no tráfego de mensagens, que poderia comprometer o sincronismo exigido pelo Delphus.

Por exemplo, se um processo é adicionado para ser monitorado com um tempo máximo de detecção de falha de 3000 ms e cada período TDMA dura 1500 ms, este processo pode ter seus *heartbeats* enviados em períodos alternados e ainda assim garantir que sua falha será detectada em no máximo 3000 ms.

Em cada fatia de *heartbeat* um módulo local envia *heartbeats* correspondentes a um sub-conjunto das suas entidades monitoráveis locais, como mostrado na Figura 14.

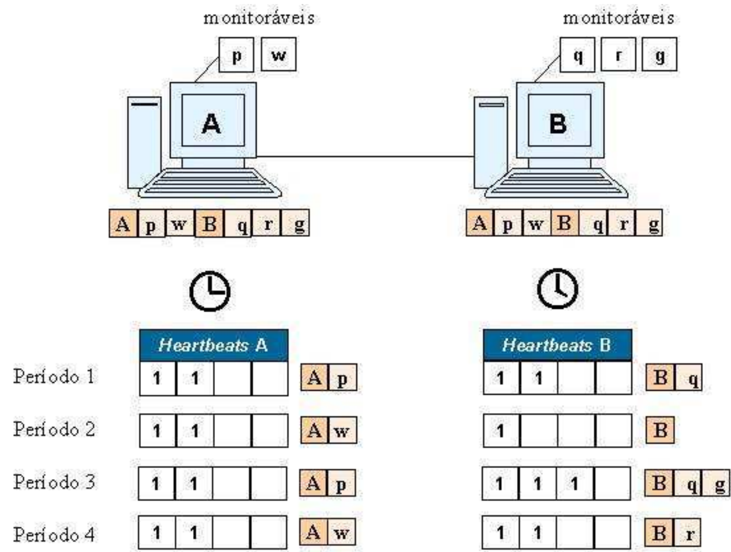


Figura 14: Sequência de Transmissões de *Heartbeats*

O envio de *heartbeats* segue uma seqüência que se repete de tempos em tempos, definindo um super período que engloba vários períodos TDMA. A Figura 15 mostra uma sequência de períodos TDMA, e os monitoráveis que foram escalonados para transmitir seus *heartbeats* em cada um. É possível verificar que a cada 6 períodos, a sequência de transmissões é repetida. Desta forma, o escalonador do Delphus precisa analisar apenas um super período para determinar em que períodos um monitorável terá seus heartbeats transmitidos.

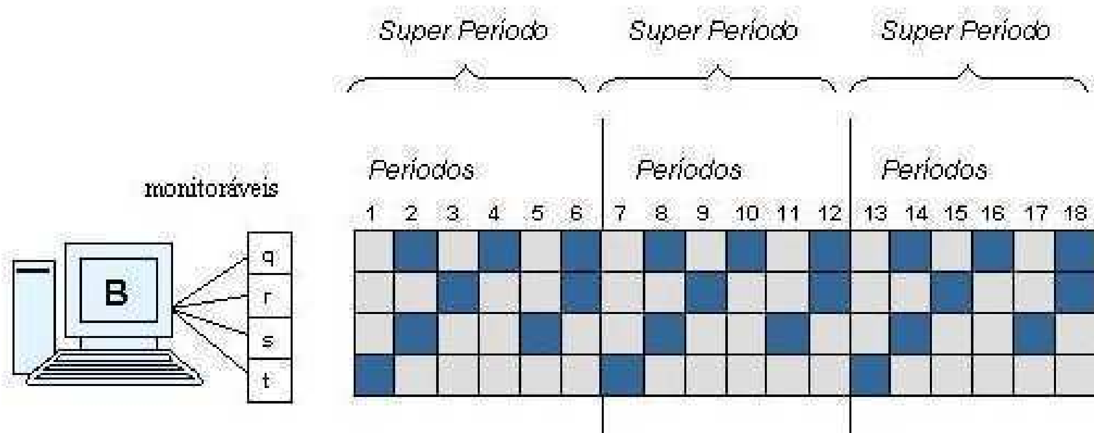


Figura 15: Super Períodos

O escalonamento de uma determinada freqüência de *heartbeats* não será possível quando não existir uma seqüência apropriada de fatias do período TDMA com sub-fatias livres.

Considere por exemplo, a seguinte situação: a seqüência de *heartbeats* de um módulo está se repetindo a cada 6 períodos do TDMA, e dentro de cada uma dessas seqüências, as fatias 1, 3 e 5 não têm sub-fatias livres; neste caso não é possível escalonar uma freqüência de 1 *heartbeat* a cada 3 períodos TDMA, como visto nas Figuras 16(a) e 16(b).

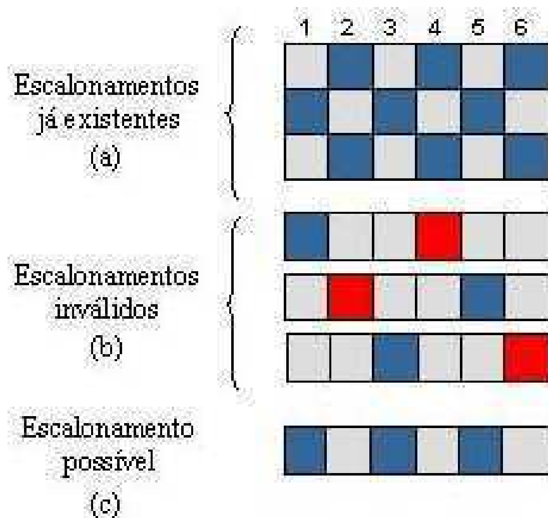


Figura 16: Ajuste de Frequência de Transmissões

Embora contra intuitivo, o aumento da freqüência de emissão de *heartbeats* para um determinado monitorável pode viabilizar o escalonamento. No exemplo anterior, não é possível escalonar uma freqüência de 1 *heartbeat* a cada 3 períodos TDMA, mas é possível escalonar uma freqüência de 1 *heartbeat* a cada 2 períodos TDMA, como visto na Figura 16(c).

Antes de requisitar ao líder do domínio a inscrição de uma nova entidade monitorável, o módulo local precisa decidir que frequência mínima de transmissão deve ser usada. Ele inicia tentando utilizar a frequência mais baixa possível, obtida pela divisão do tempo total de um período TDMA pelo tempo de detecção requisitado. Caso não seja possível encontrar uma seqüência de sub-fatias livres para a nova entidade monitorável com a frequência calculada, ele incrementa o valor da freqüência e analisa novamente. Se a frequência atingir o valor máximo (1 *heartbeat* a cada período TDMA) sem uma alternativa válida, o processo de

admissão termina e um código de erro é retornado para o processo que fez a invocação.

Observe que a duração do período TDMA define um limite mínimo para a latência de detecção (*i.e.* um limite máximo para a frequência de transmissão) que pode ser requerida por uma entidade monitorável.

Existem vários algoritmos de escalonamento de tarefas para sistemas distribuídos de tempo real. Todos eles tem o objetivo de permitir que tais tarefas tenham acesso a recursos computacionais, de comunicação ou de dados, de forma que consigam atender suas restrições temporais. O algoritmo apresentado nesta seção é classificado como um algoritmo *hard real time*, de escalonamento dinâmico, distribuído, não preemptivo [38].

Os algoritmos *hard real time* são aqueles em que os prazos para execução das tarefas devem ser garantidos em qualquer cenário do sistema. A transmissão dos *heartbeats* dos monitoráveis é um exemplo deste tipo de tarefa. A outra vertente desta classificação é a dos algoritmos *soft real time* que se destinam a tarefas onde violações esporádicas destes prazos são admitidas.

Um algoritmo de escalonamento é dito dinâmico quando a escolha das tarefas atendidas é feita durante a execução. Ele se destina a ambientes como o de execução do Delphus, onde as tarefas a serem servidas não são previamente conhecidas. O escalonamento estático só é possível de ser adotado quando as tarefas e suas propriedades são conhecidas durante as fases de projeto e desenvolvimento do software e é possível criar uma programação fixa de seu atendimento.

Os algoritmos preemptivos são aqueles que permitem que a execução de uma tarefa seja interrompida, por outra de maior prioridade. Isso não ocorre com o Delphus. A transmissão do *heartbeat* de um monitorável nunca é interrompida pela transmissão de outro monitorável.

Algoritmos como o adotado pelo Delphus são considerados distribuídos porque não são executados por uma entidade única, e sim de forma descentralizada, em cada ponto local do sistema. Cada módulo do serviço tem autonomia para escalonar transmissões de *heartbeats* dos monitoráveis locais independentemente do escalonamento dos demais módulos.

O algoritmo EDF (*Earliest Deadline First*) é um conhecido algoritmo de escalonamento *hard real time* e dinâmico [38]. No entanto, ele é um algoritmo preemptivo. Isso significa que ele permite que tarefas sejam interrompidas para a execução de outras com maior prioridade

(desde que não provoque violação de restrições temporais e de segurança). Ele é destinado a cenários onde o tempo de execução das tarefas é grande e varia de uma para outra. No caso do Delphus, o tempo em que as tarefas (transmissão de *heartbeats*) ocupam o recurso compartilhado (canal de comunicação síncrono) é muito pequeno e esta abordagem não se aplica. Entretanto, existem muitas similiaridades entre os dois algoritmos. Ambos assumem que as tarefas são independentes entre si, executadas periodicamente e que o tempo gasto com sua atividade computacional é conhecido a priori. Além disso, o EDF é baseado em prioridades dinâmicas. As tarefas possuem uma propriedade chamada prioridade que, conforme o tempo passa, tem seu valor incrementado. Periodicamente, o algoritmo seleciona as de maior prioridade para usar o recurso compartilhado. Este procedimento também é adotado pelo Delphus. Os monitoráveis possuem um contador que guarda a quantidade de períodos que faltam para que ele tenha seu *heartbeat* transferido. A proporção que o os períodos são processados pelo serviço, o valor do contador é decrementado. A cada período, os monitoráveis que possuem o contador indicando que falta apenas 1 período para que seu prazo se esgote, tem seu *heartbeat* transmitido.

Portanto, pode-se dizer que o algoritmo de escalonamento do Delphus é uma versão adaptada do EDF para um cenário simplificado de tarefas não preemptivas.

3.4 API

De modo a permitir a utilização do serviço por aplicações distribuídas, foi definida uma API (*Aplication Programming Interface*). Seu projeto seguiu algumas diretrizes básicas, apresentadas a seguir.

- As funções da API não devem conter qualquer implementação essencial dos protocolos do serviço. As funcionalidades básicas do serviço devem estar implementadas no módulo do sistema operacional e a API deve apenas fornecer acesso a elas.
- A API deve favorecer a diminuição do acoplamento entre as aplicações clientes e o serviço.
- A API deve prever futuras extensões das funcionalidades do serviço, sem que isso

implique na mudança da interface já fornecida.

Um conceito importante empregado na API do serviço é o de eventos. Ao invés de limitar o serviço à notificação exclusivamente de falhas, foi criado o conceito de notificação de eventos. Atualmente, a única informação presente nas mensagens de *heartbeat* trocadas pelos módulos do serviço é o estado dos monitoráveis (se estão corretos ou não). Em versões futuras do serviço, outras informações sobre os monitoráveis poderiam ser transmitidas nestas mensagens, tornando maior o número de eventos tratados pelo serviço. Por isso, a API do serviço permite que uma entidade se cadastre como notificável da ocorrência de eventos, e não apenas de falhas. Atualmente, apenas uma classe de eventos foi criada: eventos de monitoramento. Eles são três: admissão de monitorável, falha em monitorável e remoção de monitorável.

A seguir, vemos os tipos de funcionalidades que devem ser fornecidas pelas APIs do serviço.

3.4.1 Administração do Serviço

O serviço precisa prover um modo de ser gerenciado por usuários especiais como administradores de sistema. Desta forma, um conjunto de funções deve permitir que os módulos do serviço possam ser iniciados e interrompidos, de acordo com as necessidades do sistema. As funções deste grupo devem fornecer as seguintes funcionalidades:

- iniciação do serviço em uma máquina, e adição desta máquina como entidade monitorável;
- interrupção do serviço em uma máquina, e remoção desta máquina e de suas entidades da lista de entidades monitoráveis;
- consulta do estado do serviço em uma máquina.

3.4.2 Configuração de Monitoração

Um conjunto de funções deve permitir a configuração das entidades a serem monitoradas. As funções deste grupo devem fornecer as seguintes funcionalidades:

- cadastro de novas entidades como monitoráveis;
- remoção de entidades da monitoria;
- consulta à lista de todas as entidades monitoráveis cadastradas;
- consulta a entidades monitoráveis de uma determinada máquina;
- localização de uma entidade monitorável específica.

3.4.3 Notificação de Falha

A API deve permitir a consulta do estado das entidades monitoráveis. As funções deste grupo devem fornecer as seguintes funcionalidades:

- consulta sobre o estado de uma determinada entidade monitorável, ou seja, se ela é considerada correta ou não, e do tempo a que se refere esta informação;
- consulta à lista de todas as entidades detectadas como corretas;
- consulta à lista de todas as entidades detectadas como falhas;

A API deve também permitir formas automáticas de informar as entidades notificáveis sobre a ocorrência de eventos nas entidades monitoráveis. As funções deste grupo devem fornecer as seguintes funcionalidades:

- cadastro de entidades para serem notificadas sobre a ocorrência de eventos em entidades monitoráveis;
- remoção de entidades notificáveis;
- consulta à lista de todas as entidades notificáveis cadastradas;
- consulta à lista de todas as entidades notificáveis interessadas na ocorrência de eventos em uma entidade monitorável específica;
- localização de uma entidade notificável específica.

4 Implementação do Serviço Delphus

Uma implementação do Delphus foi feita para uma rede Ethernet com o sistema operacional Linux. Seu objetivo foi dar o primeiro passo para a implementação do serviço, produzindo uma versão preliminar, que pudesse servir como ponto de partida para os trabalhos futuros que integram o projeto [13]. Ela também teve a finalidade de demonstrar o funcionamento dos protocolos descritos no projeto do serviço, apresentados na Seção 3, e um caminho para implementá-los.

Nesta seção, explicaremos como cada elemento da arquitetura do serviço foi desenvolvido. A opção pelo Linux deve-se em primeiro lugar ao fato de que este é um sistema de código aberto. Isso facilita muito o processo de desenvolvimento e testes, ao permitir consultas a seu código fonte, o que traz maior confiança sobre seu comportamento. Além disso, este é um sistema amplamente conhecido e utilizado nos sistemas distribuídos modernos. Um serviço para o Linux, como esta implementação do Delphus, tem uma vasta base de clientes em potencial.

O código foi implementado seguindo padrões de estilo de codificação, com o objetivo de facilitar o trabalho de futuras equipes no projeto. O código em C adotou o padrão definido por Linus Torvalds [4], e o código Java, o Java Code Conventions [2], definido pela Sun Microsystems. Um outro cuidado foi tomado para facilitar trabalhos futuros: o inglês foi usado em sua codificação e documentação. Se futuramente este serviço for disponibilizado como código aberto à comunidade de desenvolvedores, esta decisão será muito bem vinda pelos profissionais de todo o mundo que se dispuserem a estudá-lo e desenvolvê-lo.

O módulo de controle do serviço foi implementado como um módulo de núcleo do Linux. Desta forma ele estende as funcionalidades do núcleo, adicionando um novo serviço ao sistema operacional. Sendo um módulo do núcleo, o serviço tem acesso às estruturas e funções internas do Linux, como se fosse parte do núcleo e permite ainda ser ativado e desativado em uma máquina, sem que ela necessite ser reiniciada.

Esta implementação considera dois tipos de entidades monitoráveis: máquinas e processos. Diferentemente de outros serviços que consideram também objetos de aplicação como entidades monitoráveis, o Delphus não os monitora diretamente. Isto se dá pelo fato de que,

não seria possível para o Delphus implementar alguma sincronia na consulta do estado dos objetos, sem que isso interferisse na política de escalonamento do sistema operacional. O estado dos objetos só poderia ser consultado quando seus processos assumissem a CPU, uma vez que sua existência é restrita à estrutura de seus processos. Então, o Delphus teria que manipular o escalonamento dos processos, para conseguir se comunicar com os objetos nos tempos certos que garantissem os níveis de qualidade de serviço assumidos. Isso certamente traria mais complexidade para garantir a semântica perfeita da detecção de falhas e para manter suas garantias de qualidade de serviço. Além do mais, isso provavelmente traria um alto impacto no desempenho do sistema, restringindo o universo de sistemas que poderiam usar o serviço.

A semântica da detecção de falhas em objetos poderia ter sido enfraquecida para que sua monitoria fosse implementada sem causar grandes interferências no funcionamento padrão do sistema. Entretanto, foi considerado que isso desvirtuaria os objetivos principais do projeto e então, foi assumido que a monitoria de um objeto deve ser feita através da monitoria de seu processo. Entretanto, como será apresentado na Seção 4.3, um objeto pode usar a API do serviço para cadastrar seu processo como monitorável, de maneira tão simples quanto se estivesse cadastrando a si mesmo, abstraindo informações como *pid*, mas na verdade, seus processos é que se tornarão entidades monitoráveis.

Dois tipos de entidades notificáveis são considerados nesta implementação: processos e objetos. Os processos são notificados através do envio de sinais do sistema operacional. Já os objetos são notificados através da invocação de métodos.

Os elementos Notificador, Coração e Escalonador, definidos no projeto do serviço, foram implementados como grupos de funções do módulo de núcleo. Algumas destas funções são invocadas indiretamente pelo usuário, via API do serviço, como por exemplo as relacionadas com o Escalonador, que adicionam e removem entidades monitoráveis. Outras funções, como a relacionada ao Coração, por exemplo, cuja função é enviar as mensagens de *heartbeat* periodicamente, são invocadas por *threads* internas do módulo, detalhadas mais a frente na Seção 4.2.1.

Nesta implementação, é considerado que um processo falha por parada quando seu identificador não faz mais parte da tabela de processos do sistema operacional ou quando seu

estado passa a ser *zombie* [11]. É considerado que uma máquina falha por parada quando é desligada ou quando o sistema operacional apresenta algum erro que interrompe o funcionamento de todos os seus serviços.

Nesta seção, os detalhes da implementação do serviço serão apresentados e discutidos. Primeiramente, o *hardware* usado para implementar a camada de comunicação do serviço será descrito na Seção 4.1. Na Seção 4.2 é descrito como os protocolos do serviço foram implementados, como as mensagens do serviço são estruturadas, como o tamanho de um período TDMA é calculado e como condições de corrida foram tratadas. Por fim, a Seção 4.3 apresenta as APIs em C e Java desenvolvidas para o serviço.

4.1 Descrição do Hardware

Esta implementação do serviço utiliza uma rede Ethernet de 100 *Mbps* como canal síncrono de comunicação. As máquinas do domínio de detecção são conectadas por uma rede extra, composta de placas de rede de 100 *Mbps*, cabos par trançado e um *switch* Fast Ethernet de 100 *Mbps*.

Em uma rede deste tipo, alguns fatores podem dificultar a obtenção do nível de sincronia e confiabilidade exigidos pelo serviço. Entretanto, muitos destes fatores ocorrem em situações de sobrecarga no tráfego da rede. Por exemplo, *buffers* de dispositivos como *switches* podem esgotar o espaço disponível de armazenamento de mensagens. Isso faz com que algumas mensagens sejam perdidas e tenham que ser retransmitidas várias vezes, até que a carga na rede diminua e permita que elas atinjam seus destinatários. Essas transmissões tornam incerto o tempo total de transmissão. Entretanto, uma vez que esta rede é de uso exclusivo do serviço, que o número máximo de mensagens a trafegá-la é limitado, que o tamanho destas mensagens é limitado a apenas 10 *bytes*, e que ela possui uma configuração super dimensionada para seu nível de utilização, é possível oferecer o nível de sincronia requerido pelo Delphus.

Vale ressaltar que esta rede deve ser instalada em um ambiente controlado, onde o nível de interferência de fatores externos como ruídos e danos físicos (inivitáveis, em qualquer canal de comunicação) seja baixo o suficiente para não comprometer o sincronismo esperado

do canal. Grande parte da confiabilidade de um canal de comunicação é obtida não pela tecnologia em que se baseia, mas pelas condições do ambiente onde ele está instalado.

É importante dizer que não há nenhuma dependência do Delphus com a tecnologia Ethernet. Seus protocolos apenas consideram a existência de um canal síncrono, que os módulos do serviço possam utilizar para comunicação. Outras tecnologias poderiam ser utilizadas. Por exemplo, uma nova placa Ethernet poderia ser desenvolvida para transmitir nos fios sobressalentes e sem uso, que existem nos cabos de par trançado tradicionais, as mensagens síncronas[3] e usar o restante dos fios do cabo para transmitir as mensagens normais da rede. Isso anularia os inconvenientes e o custo de se duplicar a rede.

Outra solução, que vem sendo estudada por Brito [13, 14] é o desenvolvimento de um dispositivo de comunicação personalizado para o Delphus. Este dispositivo implementaria algumas técnicas de tolerância a falhas, como transmissão redundante de mensagens e códigos de detecção de erros. Essas técnicas ajudariam a reduzir a probabilidade de ruídos comprometerem o sincronismo do canal.

Este dispositivo permitiria também a implementação de um componente *watchdog* em *hardware* [28], que monitoraria a transmissão de mensagens síncronas do serviço. No momento em que o serviço deixasse de enviar uma mensagem quando esperado, o componente desligaria a máquina, forçando-a a falhar. Isso preveniria o sistema contra um comportamento assíncrono do Delphus, que poderia ter consequências desastrosas. É importante lembrar que esta ação do *watchdog* seria realizada em uma situação com probabilidade muito baixa de ocorrer, visto que, como explicado nesta seção e na Seção 4.2, várias medidas foram tomadas para fazer o Delphus funcionar de forma síncrona. Ela serve para garantir completamente que o serviço não se comportará diferentemente de sua especificação, não importando quantas mudanças ocorram no restante do sistema.

[3] Os termos mensagem síncrona/assíncrona serão usados para denotar mensagens que são enviadas através da rede síncrona/assíncrona.

4.2 Descrição do Software

4.2.1 Threads do Núcleo

Um módulo local do serviço é implementado por três *threads* em um módulo de uma versão genérica do sistema operacional Linux, como mostrado na Figura 17. A *thread do_period* é responsável pela implementação das principais funcionalidades do serviço. Além disso, ela envia as mensagens na rede síncrona. A *thread sync_rec* é responsável por receber mensagens pela rede síncrona. Ela permanece a maior parte do tempo bloqueada, esperando pela chegada de mensagens no canal síncrono. Depois de receber uma mensagem, ela identifica o tipo da mensagem, e invoca uma função manipuladora específica para processá-la. Foram definidos quatro tipos de mensagens síncronas. São elas: *proposta de liderança*, *sincronização*, *anúncio* e *heartbeat*.

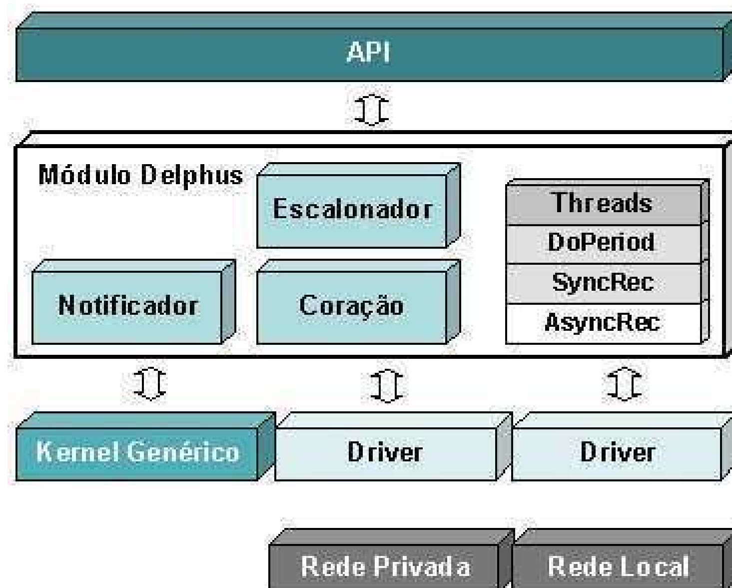


Figura 17: Threads de Núcleo

Estas duas *threads* implementam o núcleo do serviço de detecção de falhas, e são configuradas para ter prioridade de tempo real. *Threads* e processos com esta prioridade têm privilégios na política de escalonamento do Linux, e são escalonados tão logo se tornem prontos para executar.

Um problema a ser resolvido nesta implementação é a possibilidade de outros processos e *threads* serem configurados para ter a mesma prioridade das *threads* do Delphus. Isso diminuiria o tratamento privilegiado do serviço pelo escalonador e certamente interferiria em sua sincronia. No entanto, alguns fatores diminuem esta possibilidade. Em primeiro lugar, somente o super usuário tem permissão de alterar a prioridade de uma *thread* ou processo. Desta forma, somente este usuário poderia configurar outros processos e *threads* para concorrerem com o Delphus. Em segundo lugar, os valores das prioridades usados para as duas *threads* são os maiores permitidos pelo tipo *unsigned long* que é usado para armazená-los. O maior valor é atribuído à prioridade da *thread do_period*. O segundo maior valor, à *thread sync_rec*. Desta forma, mesmo que haja no sistema outros processos e *threads* que utilizem prioridade de tempo real, eles poderão utilizar uma ampla faixa de valores inferiores à usada pelo Delphus, sem que o funcionamento do serviço seja comprometido. Valores diferentes são atribuídos a cada *thread* para que, numa situação hipotética, caso as duas acordem em um mesmo instante, *do_period* seja sempre escolhida pelo escalonador.

A terceira *thread*, *async_rec*, é a versão assíncrona do *sync_rec*. Ela recebe mensagens através da rede assíncrona. Existem cinco tipos de mensagens que podem ser recebidas, elas são: *requisição de inscrição*, *requisição de remoção*, *recusa de inscrição*, *atualização* e *confirmação*. A única tarefa desta *thread* é implementar os procedimentos de inscrição e remoção, gerenciando as mensagens de anúncio que devem ser enviadas na fatia de tempo destinada aos anúncios pelo líder. Na medida que ela não está associada a prazos, esta *thread* não requer uma prioridade de tempo real.

4.2.2 Iniciação do Serviço

A iniciação do serviço é realizada através da chamada à função *start_monitoring*, que será detalhada a seguir.

O primeiro passo desta função, é validar os seus argumentos. Caso algum problema seja detectado com o formato ou faixa de valores de algum deles, um código de erro é retornado para o processo que a invocou.

Depois, a função inicia as *threads sync_rec* e *async_rec* no módulo Delphus. Então, ela

força o processo que fez a invocação a dormir por $n \times T_{periodo}$, onde n é o número máximo de máquinas que podem fazer parte de um domínio de detecção e $T_{periodo}$ é o tamanho de um período TDMA. O cálculo desta variável é descrito em detalhes na Seção 4.2.4. Esta espera tem o objetivo de aguardar o recebimento de uma mensagem de sincronização, para que seja possível identificar o líder e assim, solicitar dele a admissão no domínio de detecção.

A espera por n períodos TDMA está associada ao pior caso, onde já existem n máquinas no domínio, e n máquinas falham, em sequência, uma por período, enquanto a nova máquina está iniciando. Neste caso, a liderança do domínio vai sendo assumida pelas máquinas já existentes no domínio, seguindo sua ordem na lista de monitoráveis. Entretanto, antes que possam enviar as mensagens de líder, estas máquinas falham. Após ter passado n períodos TDMA, o domínio terá se desfeito e não existirá um líder. Neste caso, a máquina que está iniciando iniciará a formação de um novo domínio. Se após a passagem deste período de tempo, alguma máquina do domínio original restar funcionando, ela será identificada como líder do domínio pela máquina que está iniciando.

Quando o processo que invocou a função *start_monitoring* acorda, ele verifica se o líder do domínio ainda é desconhecido. Isto só acontecerá se nenhuma mensagem de sincronização for recebida pela *thread sync_rec* durante o tempo em que o processo estava dormindo. Neste momento, é possível ter certeza de que nenhum domínio de detecção foi estabelecido ainda. É possível que existam outras máquinas tentando paralelamente formar um domínio, mas nenhuma delas se considera líder ainda. Neste caso, é iniciado o protocolo para estabelecer o primeiro líder no domínio.

Isto é implementado com a máquina tentando se impor como líder do domínio, enviando uma mensagem de proposta de liderança no canal síncrono. Então, a máquina aguarda um período de tempo longo o suficiente para receber outras mensagens de proposta concorrentes de outras máquinas[4]. Este tempo foi assumido igual a $T_{periodo}$. Ele é suficiente também para receber eventuais mensagens de sincronização enviadas por um líder recém eleito, que estava dormindo, na fase final de admissão, e que acordou instantes antes da máquina em questão

[4]Este período deve ser suficiente também para que possíveis colisões de mensagens sejam resolvidas pela rede Ethernet e as mensagens entregues aos destinatários. Como o canal de comunicação síncrono é de uso controlado, este tempo é bem pequeno.

tentar se impor como líder.

Se apenas a sua proposta de liderança for ouvida, dentro do período de espera, ela se considera líder do domínio. Se mais de uma proposta de liderança for recebida, a máquina espera por um período de tempo aleatório e repete todos os passos de iniciação. A qualquer momento, se uma mensagem de sincronização for recebida, o processo de proposta de liderança é encerrado e a máquina transmissora da mensagem é assumida como líder. Se varias máquinas iniciarem o processo de iniciação em paralelo, esta espera aleatória de tempo garante que em algum momento uma delas consiga se impor como líder do domínio e então, permita a admissão das demais.

Como foi discutido anteriormente, o método de eleição de líder durante a iniciação depende da maneira com que o canal de comunicação síncrono trata colisões de mensagens. Como uma rede Ethernet foi usada nesta implementação para construir este canal, a ocorrência de possíveis colisões de mensagens é transparente para o serviço. Todas as mensagens enviadas serão recebidas, mesmo que haja colisões. As colisões provocam retransmissões das mensagens em momentos alternados, até que todas trafeguem o canal e sejam recebidas. Para o serviço é como se elas não ocorressem.

Após a descoberta do líder, se ele não for a máquina que está iniciando, um pedido de inscrição no domínio deve ser enviado para o líder. O procedimento de inscrição utiliza a rede assíncrona, conforme explicado na Seção 3. Caso o processo de inscrição seja bem sucedido[5], a *thread do_period* é criada e um código de sucesso é retornado para o processo que iniciou a requisição.

4.2.3 Execução do Serviço

A *thread do_period* executa um laço que continuamente envia mensagens síncronas nas suas fatias TDMA, conforme pode ser visto no Algoritmo 1. Antes de enviar cada mensagem, a

[5]Como visto anteriormente, o procedimento de inscrição pode falhar se um particionamento acontecer na rede assíncrona, ou se o líder corrente falhar. No primeiro caso, um erro é retornado ao processo que fez a solicitação, enquanto no último, um novo procedimento de inscrição é iniciado assim que um novo líder for eleito.

thread dorme por um tempo ϵ[6], necessário para garantir que todas as máquinas do domínio estarão sintonizadas na mesma fatia do período TDMA. Isso garante que todas as máquinas receberão as mensagens durante a fatia para a qual foram definidas. O cálculo desta e das outras variáveis de tempo referenciadas nesta seção é descrito em detalhes na Seção 4.2.4.

As mensagens enviadas dependem do módulo ser líder do domínio ou não. Desta forma, antes de enviar as mensagens sincronização e anúncio, exclusivas do líder, a *thread do_period* testa se o módulo local é o líder corrente.

[6] ϵ é o limite superior de dessincronização entre quaisquer duas máquinas de um domínio de detecção

Algoritmo 1 *Thread do_period*

enquanto $servicoIniciado = verdade$ **faça**

    //Sincronização
    $sincronizacaoRecebida = falso$
    se $maquinaCorrente = lider$ **então**
        $t_{inicio} = t_{corrente}$
        dorme até $t_{inicio} + \epsilon$
        msg.$\Delta_{prep} = t_{corrente} - t_{inicio}$
        envia *sincronizacao* $\forall maquina_i \mid maquina_i \in dominio$
    senão
        dorme até $t_{inicio} + T_{periodo} + \epsilon + T_{prep} + T_{com} + T_{rec}$

        //Tratamento de falhas no líder
        se $sincronizacaoRecebida = falso$ **então**
            $lider = lider.proximo$
            $t_{inicio} = t_{inicio} + T_{periodo}$
        fim se
    fim se
    dorme até $t_{inicio} + T_{fatia}$

    //Anúncio
    se $maquinaCorrente = lider$ **então**
        dorme até $t_{inicio} + T_{fatia} + \epsilon$
        se $admissaoPendente \neq \varnothing$ **então**
            envia *anuncio de admissao* $\forall maquina_i \mid maquina_i \in dominio$
        senão
            se $remocaoPendente \neq \varnothing$ **então**
                envia *anuncio de remocao* $\forall maquina_i \mid maquina_i \in dominio$
            fim se
        fim se
    fim se
    dorme até $t_{inicio} + 2 \times T_{fatia}$

    //Transmissão de heartbeats
    $fatiasHbtUsadas = 0$
    paratodos $maquina_i \in dominio$ **faça**
        $heartbeatRecebido = falso$
        $transmissorHeartbeats = maquina_i$
        dorme até $t_{inicio} + (2 + fatiasHbtUsadas) \times T_{fatia} + \epsilon$
        se $maquina_i = maquinaCorrente$ **então**
            consulta estado dos monitoráveis locais
            envia *heartbeats* $\forall maquina_j \mid maquina_j \in dominio$
            dorme por T_{rec}
        senão
            dorme por $T_{prep} + T_{com} + T_{rec}$
            se $heartbeatRecebido = falso \wedge maquina.estado = correto$ **então**
                $maquina.estado = falho$
                notifica falha às entidades notificáveis da máquina
            fim se
        fim se
        $fatiasHbtUsadas = fatiasHbtUsadas + 1$
        dorme até $t_{inicio} + (2 + fatiasHbtUsadas) \times T_{fatia}$
    fim para
fim enquanto

Sincronização

Uma mensagem de sincronização contém o número do período corrente, as propriedades do líder (seu identificador e seu endereço IP na rede assíncrona) e o valor Δ_{prep}, que é o tempo decorrido entre o início do período no líder e o preparo da mensagem de sincronização. O tempo Δ_{prep} é usado pelos demais módulos para calcular o horário de início do período.

Após o líder enviar esta mensagem, todos os módulos que executam a *thread do_period* dormem pelo tempo necessário para que a mensagem chegue aos destinatários e seja devidamente processada por eles. Este tempo engloba o tempo gasto para o preparo, transmissão e recebimento da mensagem: T_{prep}, T_{com} e T_{rec}, respectivamente.

Tratamento de Falhas no Líder

Em seguida, é testado se a mensagem de sincronização foi recebida, verificando se a variável boleana *sincronizacaoRecebida* contém o valor *verdadeiro*. Esta variável inicia o período com valor igual a *falso* e é alterada no recebimento de uma mensagem de sincronização pela *thread sync_rec*, vista no Algoritmo 2 explicado mais adiante. Caso esta variável indique que a mensagem de sincronização não foi recebida, a próxima máquina da lista de máquinas do domínio de detecção é escolhida como nova líder, seguindo a ordem em que as máquinas se inscreveram. Outro procedimento é calcular o horário de início do período somando o comprimento de um período ($T_{periodo}$) ao horário de início do período passado.

Anúncio

Em seguida, a *thread do_period* executando no líder passa a executar a fatia de anúncio. Inicialmente, realiza um teste para saber se é o líder do domínio, como na fatia de sincronização. Caso seja, testa se a variável *admissaoPendente* referencia algum monitorável que deva ser admitido no domínio. Em caso positivo, envia uma mensagem de anúncio de admissão, que contém as propriedades do monitorável a ser incluído nas listas de monitoráveis de cada módulo do serviço. Caso esteja nula a variável *admissaoPendente*, testa o conteúdo da variável *remocaoPendente*. Se não estiver nula, envia uma mensagem de anúncio de remoção, que contém apenas o identificador do monitorável a ser excluído das

Algoritmo 2 *Thread sync_rec*

```
enquanto servicoIniciado = verdade faça
  espera recebimento de mensagem msg

  se msg.tipo = sincronizacao então
    sincronizacaoRecebida = verdade
    periodoCorrente = msg.periodo
    lider = msg.lider

    //Cálculo do horário de início do período
    t_inicio = t_corrente − T_sinc − msg.Δ_prep
  senão

    se msg.tipo = admissao então
      maquina = maq | ∃ maq ∈ dominio ∧ maq.id = msg.idMaquina
      se maquina = ∅ então
        dominio = dominio ∪ {msg.monitoravel}
      senão
        maquina.monitoraveis = maquina.monitoraveis ∪ {msg.monitoravel}
      fim se
      notifica entidades notificáveis da ocorrência de admissão
    senão

      se msg.tipo = remocao então
        maquina = maq | ∃ maq ∈ dominio ∧ maq.id = msg.idMaquina
        se maquina ≠ ∅ então
          dominio = dominio − {maquina}
        senão
          maquina.monitoraveis = maquina.monitoraveis − {msg.monitoravel}
        fim se
        notifica entidades notificáveis da ocorrência de remoção
      senão

        se msg.tipo = heartbeat então
          heartbeatRecebido = verdade
          atualiza estado dos monitoráveis de transmissorHeartbeats
          notifica entidades notificáveis da ocorrência de falhas, se houverem
        senão

          se msg.tipo = proposta de lideranca então
            lider = msg.lider
          fim se
        fim se
      fim se
    fim se
  fim se
fim enquanto
```

listas de monitoráveis. Apenas uma das duas variáveis ou nenhuma delas estará preenchida em cada período. O último passo desta fase é dormir até o final do tempo reservado para uma fatia TDMA e é executado por todas as *threads*, líder ou não.

Transmissão de Heartbeats

A última etapa seguida por *do_period* é executar um laço que percorre todas as máquinas do domínio. Para cada uma delas, é testado se possui o mesmo identificador da máquina corrente. Caso positivo, a *thread* assume aquele como sua fatia de transmissão de *heartbeat*, consulta o estado dos processos monitoráveis programados para transmitir naquele período e envia sua mensagem de *heartbeats*.

O passo seguinte para qualquer máquina, é dormir pelo tempo necessário para que a mensagem seja recebida por todas as máquinas da rede e devidamente processada, conforme ocorre na fatia de sincronização. Antes que o tempo da fatia se esgote, é testado se a mensagem de *heartbeat* da máquina foi recebida ou não. Se não, a máquina e seus processos são sinalizados como falhos, e os processos notificáveis são notificados da ocorrência com o envio de sinais.

A *thread sync_rec* é responsável por receber as mensagens síncronas e invocar funções auxiliares específicas para cada uma, que por sua vez interpretam as mensagens. Ela complementa a *thread do_period* executando todo o processamento que ocorre nos destinatários das mensagens síncronas de cada fatia. O Algoritmo 2 apresenta a lógica desta *thread* e de suas funções auxiliares.

Sempre que uma mensagem de sincronização é recebida, as variáveis *períodoCorrente*, *sincronizacaoRecebida* e t_{inicio} são atualizadas. A variável *periodoCorrente* serve como um contador que guarda o número de ordem do período corrente. Ela é útil durante o processo de inscrição de monitoráveis, explicado na Seção 3.3.2. A variável t_{inicio} é utilizada para controlar a duração das fatias do período TDMA, conforme visto anteriormente. Seu cálculo é feito através da subtração no horário corrente da máquina, do erro máximo de sincronização T_{sinc} e do tempo Δ_{prep} enviado pelo líder na mensagem de sincronização. Este cálculo resulta no tempo aproximado em que o líder iniciou o período corrente. Por fim, a variável *sincronizacaoRecebida* serve para indicar à *thread do_period* que a sincronização ocorreu, e

portanto o líder não falhou.

Ao receber uma mensagem de anúncio de admissão, a *thread sync_rec* procura por uma máquina com o identificador de máquina vindo na mensagem. Caso encontre, assume que o monitorável é um processo e o adiciona na lista de processos monitoráveis da máquina encontrada. Caso não exista máquina com o identificador de máquina informado, assume que o monitorável é uma máquina e o adiciona na lista de máquinas do domínio. A última ação é notificar o evento de admissão às entidades notificáveis, através do envio de sinais.

O tratamento de uma mensagem de anúncio de remoção é mais simples. A *thread sync_rec* procura uma máquina. Se for encontrada, remove-a da lista domínio. Se não for encontrada, procura por um processo monitorável com tal identificação e o remove da lista de monitoráveis de sua máquina. A última ação é notificar o evento de remoção às entidades notificáveis, através do envio de sinais.

Quando uma mensagem de *heartbeat* é recebida, a variável boleana *heartbeatRecebido* é atualizada para o valor verdadeiro. Ela serve para indicar à *thread do_period* que uma determinada máquina transmitiu seus *heartbeats* e que portanto está operante. Depois, os *heartbeats* são consultados e relacionados aos monitoráveis da máquina transmissora, identificada pela variável *transmissorHeartbeats*, atualizada por *do_period*. Se algum dos *heartbeats* indicar uma falha no monitorável, seu estado é atualizado para *falho* e as entidades notificáveis são notificadas da falha, através do envio de sinais.

4.2.4 Cálculo do Tamanho do Período TDMA

Crucial para o funcionamento correto do serviço é garantir que, a qualquer momento, apenas uma máquina transmitirá informações através da rede síncrona. Logo, definir precisamente o tamanho de cada fatia de tempo no período TDMA, e os instantes na fatia onde cada etapa da comunicação entre os módulos deverá ser realizada são aspectos fundamentais na implementação do serviço.

Para calcular o tamanho da fatia de tempo TDMA, diversas variáveis precisam ser consideradas. Algumas delas são fontes de incerteza que podem invalidar o cálculo, se não forem apropriadamente consideradas.

Algumas variáveis consideram um tempo de utilização da CPU pelo serviço, que pode ser inflado pela ocorrência de interrupções na máquina. As rotinas de tratamento de interrupção têm prioridade de execução na CPU e suspendem a execução de qualquer processo, ou *thread*, mesmo aqueles com prioridade de tempo real. Por este motivo, foram criadas variáveis auxiliares para representar este atraso no processamento. Estas variáveis são identificadas por terem o nome começado com a letra α.

A seguir, todas as variáveis são definidas, seguindo a ordem de interdependência entre elas. Por fim, alguns experimentos usados como fonte de informação para o cálculo são descritos e os cálculos do tamanho de uma fatia TDMA e de um período TDMA são realizados.

Tempo de um período TDMA

O comprimento de um período TDMA é definido como $T_{periodo}$, e calculado pela Fórmula 1.

$$\boxed{T_{periodo} = (n + 2) \times T_{fatia}} \quad (1)$$

O termo (n + 2) refere-se ao total de fatias existentes em um período TDMA, onde n é o número máximo de máquinas do domínio. Como visto anteriormente, o período é composto de n fatias de transmissão de *heartbeats*, uma de sincronização e outra de anúncio. A variável T_{fatia} é o comprimento de uma fatia TDMA, definido a seguir.

Tempo de uma fatia TDMA

O tamanho de uma fatia de tempo do protocolo TDMA é dado pela Fórmula 2. A Figura 18 mostra a composição de uma fatia TDMA.

$$\boxed{T_{fatia} = 2 \times \epsilon + T_{ativ}} \quad (2)$$

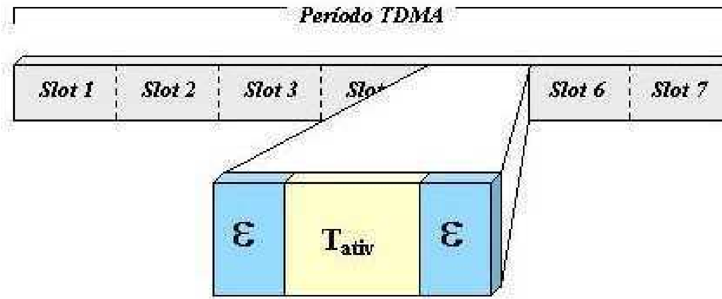


Figura 18: Composição de uma Fatia TDMA

Duas variáveis principais compõem uma fatia TDMA. A variável ϵ equivale ao tempo que a *thread do_period* deverá dormir no início e final de cada fatia, para evitar que uma mensagem seja enviada ou recebida fora da fatia a que se refere. A variável T_{ativ} é o tempo em que a *thread do_period* estará em atividade, preparando, enviando, recebendo e interpretando a mensagem da fatia. A seguir, estas variáveis são explicadas com maiores detalhes.

Tempo de atividade em uma fatia TDMA

O tempo em que a *thread do_period* tem acesso à CPU e realiza suas principais atividades é dividido em quatro fases, representadas pelas variáveis T_{prep}, T_{com}, T_{rec} e T_{trat}, como visto na Fórmula 3.

$$\boxed{T_{ativ} = T_{prep} + T_{com} + T_{rec} + T_{trat}} \quad (3)$$

A seguir, a definição de cada uma das quatro variáveis.

Tempo de preparo de mensagem

Assim que a *thread do_period* acorda do tempo ϵ na fatia TDMA, para realizar uma transmissão de mensagem, existe um processamento inicial necessário para que uma mensagem seja criada. Este tempo é dado pela Fórmula 4, composta dos seguintes termos:

- T_{esc} – tempo máximo que leva para o escalonamento da *thread* ocorrer, antes que este processamento seja iniciado;

- $T_{execprep}$ – tempo máximo necessário para executar o código que cria a mensagem, até antes da invocação da função *sock_sendmsg*, que transfere a mensagem para o dispositivo de comunicação;

- α_{prep} – atraso sobre $T_{execprep}$, causado por tratamento de interrupções.

$$\boxed{T_{prep} = T_{esc} + T_{execprep} + \alpha_{prep}} \tag{4}$$

Tempo de escalonamento

A variável T_{esc} considera o maior atraso que a *thread do_period* poderá sofrer em sua execução por causa da espera do acionamento do escalonador de processos. Este atraso máximo ocorre na situação descrita a seguir.

Em um determinado instante, ocorreria uma interrupção de relógio e a *thread do_period* teria dormido por quase todo o tempo que deveria. Faltando uma fração de tempo muito pequena para a *thread* ter seu estado mudado de *dormindo* para *pronta para execução*, o escalonador escolheria outro processo para ter acesso à CPU. Como a *thread do_period* estaria dormindo naquele instante, ela não poderia ser escolhida. Desta forma, *do_period* teria de esperar até que a próxima interrupção de relógio provocasse a mudança de seu estado. Esta mudança provocaria o acionamento do escalonador e permitiria ser escolhida por ele para ter acesso à CPU.

O valor máximo de T_{esc} será definido pela frequência de interrupções de relógio da máquina. Na situação descrita acima, a *thread* teria de esperar pelo tempo decorrido entre duas interrupções de relógio. Desta forma, o intervalo de tempo entre uma interrupção e outra define o valor de T_{esc}.

Tempo de comunicação

Outro importante fator para o cálculo do tamanho da fatia é o tempo de comunicação entre os módulos do serviço, definido como T_{com}. Ele decorre entre o instante em que um módulo inicia a transferência de uma mensagem para o dispositivo de comunicação e o instante em que esta mensagem se torna disponível para uso pelo módulo destino. Nesta

implementação, o tempo de comunicação T_{com} é dado pela Fórmula 5, composta dos seguintes termos:

- T_{transf} – o tempo máximo necessário para que uma mensagem seja transferida para o *buffer* do dispositivo de comunicação, começando assim que a função *sock_sendmsg* é invocada;

- T_{transm} – o tempo máximo necessário para que o dispositivo de comunicação transmita uma mensagem de no máximo 10 *bytes* pelo canal síncrono;

- $T_{entrega}$ – o tempo máximo decorrido entre o recebimento de uma interrupção do dispositivo de comunicação, indicando a chegada da mensagem e o instante em que a função *sock_receivemsg* retorna a mensagem na *thread sync_rec*;

- α_{transf} – o tempo de atraso para concluir a transferência da mensagem, causado por tratamento de interrupções na máquina origem;

- $\alpha_{entrega}$ – o tempo de atraso para concluir a entrega da mensagem, causado por tratamento de interrupções na máquina destino.

$$\boxed{T_{com} = T_{transf} + T_{transm} + T_{entrega} + \alpha_{transf} + \alpha_{entrega}} \quad (5)$$

Tempo de recebimento de mensagem

Após o tempo T_{com}, a mensagem estará disponível para ser acessada por um módulo. Existe um tempo T_{rec}, necessário para que ela seja decomposta e sua informação devidamente processada. Este tempo é dado pela Fórmula 6, composta dos seguintes termos:

- $T_{execrec}$ – tempo máximo necessário para executar o código que processa a mensagem;
- α_{rec} – atraso sobre $T_{execrec}$, causado por tratamento de interrupções.

$$\boxed{T_{rec} = T_{execrec} + \alpha_{rec}} \tag{6}$$

Como visto anteriormente, a *thread sync_rec* permanece bloqueada até que uma mensagem seja entregue a ela e uma interrupção do dispositivo de comunicação mude seu estado para pronta para execução. Quando isso ocorre, o escalonador é acionado. Ele dá a ela a chance de usar a CPU, visto que ela tem prioridade de tempo real, e a *thread do_period* estará dormindo.

Tempo de tratamento do não recebimento da mensagem

Depois que uma mensagem é enviada em uma fatia TDMA, todas as *threads do_period* dormem até que tenha passado o tempo necessário para que ela seja recebida e processada por todos os destinatários. Passado esse tempo, a depender do tipo da fatia, ela precisa verificar se a mensagem esperada foi devidamente recebida. Caso contrário, precisa realizar algum processamento específico, como por exemplo, notificar a falha em uma máquina, caso sua mensagem de *heartbeat* não seja recebida em sua fatia TDMA. Este tempo, é calculado a partir da Fórmula 7, composta dos seguintes fatores:

- T_{esc} – tempo máximo que leva para o escalonamento da *thread do_period* ocorrer, antes que este processamento seja iniciado;
- $T_{exectrat}$ – tempo máximo necessário para executar o código de tratamento do não recebimento da mensagem;
- α_{trat} – atraso sobre $T_{exectrat}$, causado por tratamento de interrupções.

$$\boxed{T_{trat} = T_{esc} + T_{exectrat} + \alpha_{trat}} \qquad (7)$$

Tempo para sintonia das *threads* em uma fatia TDMA

O tempo ϵ equivale ao tempo em que a *thread do_period* deve dormir no início e no final de cada fatia TDMA, para garantir que:

- os módulos mais adiantados no tempo esperem os mais atrasados atingir a mesma fatia, antes de enviar qualquer mensagem;
- os módulos mais atrasados terminem de realizar suas atividades antes que os outros entrem na fatia seguinte.

Seu valor é dado pela Fórmula 8. Ela é composta pela soma da diferença de tempo máxima entre o início de período dos módulos em relação ao líder, e do tempo máximo acumulado de desvio de relógios. Estas variáveis são descritas em detalhes a seguir.

$$\boxed{\epsilon = T_{sinc} + T_{desvio}} \qquad (8)$$

Tempo de sincronia de períodos

Nesta subseção, será apresentada a variável T_{sinc}, que é o máximo valor a que pode chegar a diferença entre os horários de início de período do líder e dos outros módulos do domínio. As fórmulas para este cálculo serão construídas ao longo da subseção, e apenas no final, suas versões finais serão apresentadas.

Conforme explicado nas seções anteriores e ilustrado na Figura 19, assim que o líder inicia uma fatia TDMA de sincronização, no instante i_1, ele marca o início do período, registrando em sua variável local t_{inicio} o horário corrente de seu relógio. Depois, o líder envia uma mensagem de sincronização para os demais módulos do domínio. Quando estes módulos recebem a mensagem, no instante i_2, eles calculam t_{inicio}, usando a Fórmula 9, subtraindo do seu horário corrente um valor τ que representa o valor aproximado do tempo decorrido entre os instantes i_1 e i_2. Este procedimento é necessário para sincronizar os inícios de período dos vários módulos do serviço.

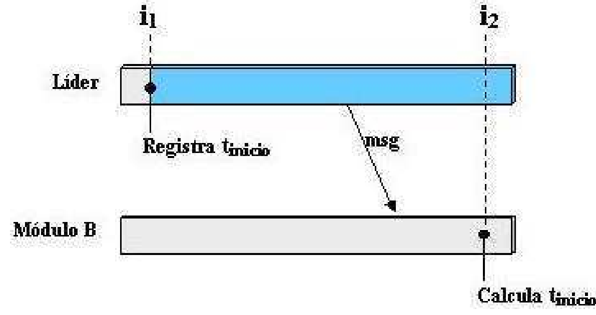


Figura 19: Início do período do líder e sincronização de período nos demais módulos

$$t_{inicio} = t_{corrente} - \tau \qquad (9)$$

No relógio de um observador externo, o instante i_1 ocorreria no horário t_i, e o instante i_2 ocorreria no horário t_r. Sendo τ_{min} o menor valor possível para a diferença $(t_r - t_i)$ e τ_{max} o maior valor, τ poderia ser calculado pela Fórmula 10. Ele seria a média do tempo necessário para ocorrer a sincronização dos períodos e por isso, poderia ser usado na Fórmula 9 para calcular o t_{inicio} de um módulo.

$$\tau = (\tau_{max} + \tau_{min})/2 \qquad (10)$$

É necessário considerar que os relógios das máquinas contam o tempo em velocidades diferentes. Esta diferença existe devido à tecnologia baseada no uso de cristais de quartzo, que é adotada.

Submetidos a uma diferença de tensão entre suas extremidades, os cristais dos relógios vibram. Um aparelho registra seus movimentos cadenciados, e assim, mede a passagem do tempo. A frequência de vibração varia de um cristal para outro, causando a diferença de velocidade com que os relógios contam o tempo. Entretanto, há uma frequência considerada média e uma proporção máxima de variação para mais ou para menos, a que pode chegar a frequência de um cristal. Esta proporção é conhecida como ρ, e deve ser considerada no cálculo de τ. Deste modo, a Fórmula 10 deveria ser modificada para a Fórmula 11, para considerar o desvio dos relógios.

$$\tau = ((\tau_{max} \times (1 + \rho)) + (\tau_{min} \times (1 - \rho)))/2 \tag{11}$$

O termo $(1 + \rho)$ serve para aumentar ainda mais o valor de τ_{max}, considerando que o maior atraso possível de relógio. O termo $(1 - \rho)$ serve para diminuir ainda mais o valor de τ_{min}, considerando a maior antecipação possível de relógio.

Resta definir como os valores τ_{max} e τ_{min} seriam calculados. Como mostrado na Figura 20, entre os instantes de tempo i_1 e i_2 deveria ocorrer o tempo ϵ, a preparação da mensagem de sincronização, seu envio pela rede e seu recebimento pelos módulos destinos. Desta forma, τ_{max} seria a soma de todas as variáveis envolvidas nestas atividades, como mostrado pela Fórmula 12.

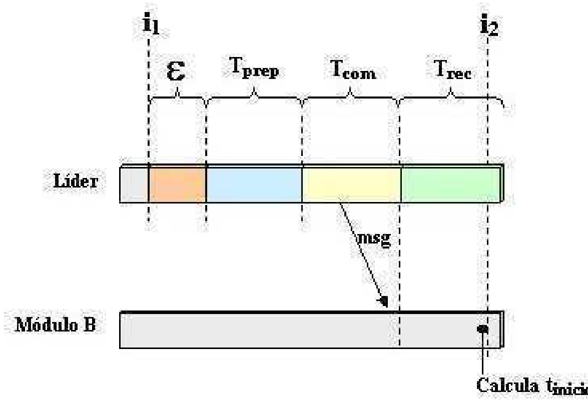


Figura 20: Composição de τ

O valor τ_{max} seria observado numa situação de carga do sistema em que o processamento seja bastante atrasado pelo tratamento de interrupções, que a *thread do_period* demore o máximo valor de T_{esc} para ser escalonada e que a mensagem demore o máximo para percorrer o canal de comunicação síncrono.

$$\tau_{max} = \epsilon + T_{prep} + T_{com} + T_{rec} \tag{12}$$

Por outro lado, τ_{min} deveria ser a soma das mesmas variáveis, considerando os menores valores que poderiam assumir. τ_{min} seria observado numa situação de pouca carga do sistema, em que ocorresse o mínimo de interrupções, o escalonamento da *thread do_period* acontecesse

assim que ela se tornasse pronta para executar, e que a mensagem percorresse o canal de comunicação com a máxima velocidade possível.

Neste cenário, τ_{min} seria muito próximo de ϵ, já que a soma $T_{prep} + T_{com} + T_{rec}$ seria desprezível. As diferenças de velocidades de transmissão de mensagens no canal síncrono são irrisórias, visto que em uma rede Ethernet, ela é próxima da velocidade da luz [43] e que a distância máxima entre duas máquinas em uma rede local é de no máximo 1000 m [44]. Além do mais, grande parte do tempo de comunicação T_{com} se deve ao atraso causado pelo tratamento de interrupções. No cenário onde ocorra apenas as interrupções de relógio e da placa de rede, o atraso é da ordem de microsegundos. Estes fatores tornam os valores das variáveis T_{prep}, T_{com} e T_{rec} próximos de 0 ms e portanto, de pouco impacto para o cálculo de τ_{min}. Portanto, τ_{min} é considerado igual a ϵ.

Há uma forma de simplificar o cálculo de τ_{min} e τ_{max}. Segundo mostra a Figura 21, o instante i_1 poderia ocorrer após a *thread do_period* do líder acordar do tempo ϵ. Desta forma, a variável ϵ não seria computada nas Fórmulas de cálculo de τ_{max} e τ_{min}, simplificando a fórmula de τ para a Fórmula 13.

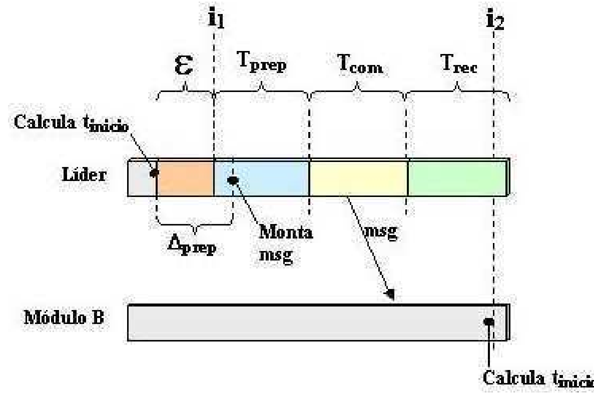


Figura 21: Otimização do cálculo de τ

$$\tau = ((T_{prep} + T_{com} + T_{rec}) \times (1 + \rho))/2 \quad (13)$$

Para considerar o espaço de tempo Δ_{prep}, decorrido entre o início do período do líder e o novo instante i_1, a mensagem de sincronização seria preparada para transportar Δ_{prep} para

os demais módulos. Estes módulos usariam então este valor na Fórmula 14 para calcular seu horário de início de período t_{inicio}.

$$\boxed{t_{inicio} = t_{corrente} - \tau - \Delta_{prep}} \tag{14}$$

Depois desta sincronização, restará ainda uma diferença de tempo entre os inícios de período dos módulos. A Figura 22 mostra que, para um observador externo, os inícios dos períodos terão uma diferença entre si. Este erro de sincronização deve-se ao fato de que τ é um valor médio, e como tal, serve para aproximar os inícios dos períodos, mas não garante que eles ficarão perfeitamente emparelhados.

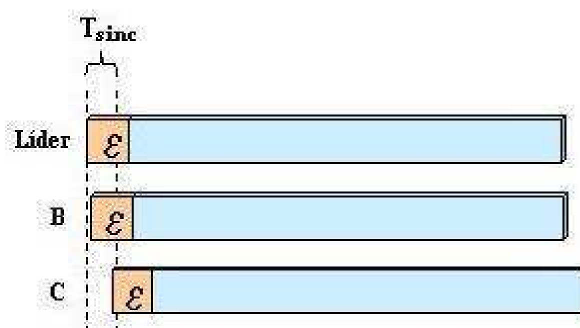


Figura 22: Erro de sincronização dos períodos

Os valores reais das variáveis T_{prep}, T_{com} e T_{rec} durante a execução do serviço é que determinarão o quão distantes os inícios dos períodos ficarão entre si. Entretanto, o uso de um valor médio para τ faz com que o erro máximo seja igual a esta mesma média, no pior caso.

Usando o exemplo da Figura 22, supondo $\tau_{min} = 0\ ms$ e $\tau_{max} = 10\ ms$, τ seria de aproximadamente 5 ms. No caso de durante a execução do módulo B, o valor real τ' ser igual a 10, a distância que B ficaria do líder seria igual a 5 ms. Se durante a execução de um módulo C, τ' fosse igual a 0 ms, a distância do módulo C seria também de 5 ms.

Deste modo, pode-se afirmar, que T_{sinc}, o erro máximo de sincronização dos períodos, tem a mesma fórmula de cálculo que τ, como mostrado na Fórmula 15, ou mais resumidamente na Fórmula 16.

$$T_{sinc} = ((\tau_{max} \times (1 + \rho)) + (\tau_{min} \times (1 - \rho)))/2 \quad (15)$$

$$T_{sinc} = ((T_{prep} + T_{com} + T_{rec}) \times (1 + \rho))/2 \quad (16)$$

Analisando a Fórmula 15, fica mais evidente o benefício de não se computar o valor ϵ no cálculo de τ. Sua ausência na composição de τ_{max} torna o erro de sincronização T_{sinc} menor, favorecendo uma economia de espaço em T_{fatia}.

Desvio acumulado de relógios

A fórmula de cálculo de T_{sinc} considera a influência do desvio dos relógios no tamanho de uma fatia TDMA. Entretanto, T_{sinc} considera o desvio máximo ocorrido em apenas um período TDMA. A cada vez que os módulos do domínio calculam o horário de início de um período (t_{inicio}), o desvio ocorrido no período passado é anulado e substituído por um novo. Eles não se acumulam quando as mensagens de sincronização são recebidas regularmente e o t_{inicio} é sempre recalculado.

O desvio somente é acumulado de um período para outro, quando o envio das mensagens de sincronização é interrompido. Neste caso, os módulos atualizam o t_{inicio} baseando-se na última mensagem recebida, e não no recebimento da mensagem do líder. Desta forma, o desvio de seu relógio se acumula de um período para outro. Este acúmulo tem que ser considerado em ϵ, para evitar que uma máquina envie uma mensagem em uma fatia TDMA quando outra máquina com relógio mais veloz julgasse já ser o início da próxima fatia.

O cálculo do desvio máximo dos relógios é feito considerando o pior caso de acúmulo de desvios. Esta situação ocorre se existir o número máximo de n máquinas no domínio e $(n-2)$ falharem em sequência, uma por período, antes de enviar a mensagem de sincronização. Neste caso, a última máquina levará $(n-2)$ períodos sem receber uma mensagem de sincronização. O desvio dos relógios das duas máquinas restantes será acumulado até que a penúltima máquina assuma a liderança do domínio e envie uma mensagem de sincronização. O cálculo deste desvio acumulado é feito a partir da Fórmula 17.

$$\boxed{T_{desvio} = (n-2) \times (n+2) \times T_{fatia} \times (2\rho)} \quad (17)$$

O termo $(n-2)$ refere-se ao pior caso, quando $n-2$ máquinas falham, e as máquinas restantes foram as últimas a unir-se ao domínio de detecção. O termo $(n+2)$ representa o número de fatias de tempo no período TDMA (n fatias de tempo para *heartbeats*, mais duas para sincronização e anúncio), e T_{fatia} é a duração de uma fatia de tempo TDMA. Logo, o termo $(n+2) \times T_{fatia}$ refere-se ao tempo de um período TDMA. O termo 2ρ refere-se ao pior caso de acúmulo de desvio, onde o relógio de uma máquina é o mais rápido possível e o da outra, o mais lento possível. Sendo assim, seus registros do tempo distanciam-se um do outro a uma taxa igual a 2ρ.

Considerações sobre o cálculo

Como visto anteriormente, o cálculo das variáveis T_{prep}, T_{com}, T_{rec} e T_{trat} depende da quantidade de interrupções que ocorrem na máquina. Como não existe um limite de interrupções que podem ocorrer, foi adotada a estratégia de se realizar experimentos em um sistema lento, de configuração pouco potente, extremamente carregado e tomar os maiores valores registrados para as variáveis em um longo espaço de tempo. De posse desses valores, o passo seguinte foi majorá-los para criar uma margem de segurança, e adotar os valores resultantes como valores máximos das quatro variáveis.

É claro que desta forma, sempre existirá a possibilidade de um sistema real apresentar uma quantidade de interrupções superior ao observado nestes experimentos. Entretanto, a solução para este problema não seria considerar uma taxa maior de interrupções no cálculo das variáveis. Sempre haveria a possibilidade de uma taxa ainda maior ocorrer.

Esta é uma questão que depende dos níveis mínimos de cobertura e latência de detecção de falhas que se pretende do serviço de detecção. Quanto maiores forem os valores adotados para as variáveis T_{prep}, T_{com}, T_{rec} e T_{trat}, maior será a cobertura do serviço, ou seja, maior a taxa de interrupções que o sistema poderá apresentar, sem que o sincronismo do serviço seja afetado. No entanto, quanto menores estes valores, maior também será o comprimento de um período ($T_{periodo}$), que define a latência mínima de detecção de falhas.

Por outro lado, quanto menores os valores adotados para as variáveis T_{prep}, T_{com}, T_{rec} e T_{trat}, menor a latência mínima de detecção de falhas. Entretanto, isto implica também em uma diminuição da cobertura do serviço.

Devido à possibilidade, mesmo que pequena, de uma taxa de interrupções maior do que a considerada nestes cálculos forçar o serviço a não se comportar de forma síncrona, um cuidado especial foi adotado no desenvolvimento das *threads* do serviço. Antes que qualquer mensagem seja enviada, é testado se a fatia TDMA corrente é a apropriada para aquela mensagem. Caso contrário, um *abnormal error* (descrito na Seção 4.3.1), é produzido e a máquina é forçada a falhar. Este tratamento almeja implementar um componente *watchdog* em *software* [28], que impeça o serviço de violar seus requisitos de sincronia.

Entretanto, a possibilidade, mesmo que remota, de uma rajada de interrupções forçá-lo a falhar ainda existirá. Se a rajada ocorrer, por exemplo, durante a execução da função *sock_sendmsg*, uma mensagem poderá ser enviada na próxima fatia TDMA. E não há como impedir este atraso, visto que ele ocorrerá fora do código do serviço. A solução para isso está em adotar o *watchdog* como um componente de hardware externo e independente. Este é um dos objetivos do trabalho de Brito [13, 14], que também é parte do projeto Delphus.

O tratamento de *watchdog* em *software* previne o sistema de falhas temporais no Delphus, mas pode afetar a disponibilidade das máquinas, caso estas falhas ocorram com frequência. Sendo assim, tem-se uma questão que depende do nível de disponibilidade das máquinas que o usuário do serviço pretende garantir no sistema. Quanto maior a disponibilidade a ser garantida, maior terá de ser o comprimento de um período TDMA, para minimizar a possibilidade de ocorrência de falhas temporais que fariam o *watchdog* forçar a máquina a falhar. Quanto menor o período TDMA, maior esta possibilidade.

Experimentos

Dois experimentos foram realizados. O primeiro teve o objetivo de medir valores para o tempo de comunicação T_{com}. O segundo, teve o objetivo de medir valores das demais variáveis que compõe uma fatia TDMA: T_{prep}, T_{rec} e T_{trat}.

No primeiro experimento, dois programas especialmente desenvolvidos foram usados. Executando em máquinas diferentes, o programa *Rx*, esperava uma mensagem de 10 bytes,

enviada pelo programa *Tx*.

O *Rx* executava em uma máquina Pentium, clock de 240 *Mhz*, com 32 *Mbytes* de memória RAM, disco rígido de 20 *Gbytes*, conectado à rede do laboratório por placas de rede Ethernet de 100 *Mbps*, com a distribuição Linux Mandrake 8.2, kernel 2.4.18. O programa *Tx* executava em uma máquina similar, com clock de 700 *Mhz* e 128 *Mbytes* de memória RAM. Estas eram as máquinas mais lentas e de configuração menos potente existentes no laboratório. O *Rx* foi configurado para ter a maior prioridade de escalonamento de sua máquina e invocava a função de recebimento de mensagem pelo modo não bloqueante, repetidas vezes, dentro de um laço.

Na primeira etapa do experimento, foi realizada a medição de T_{com} com os sistemas sem muita carga e portanto, gerando poucas interrupções.

Na segunda etapa, as medições ocorreram em sistemas sobrecarregados. Com a intenção de produzir um aumento da taxa de todos os tipos de interrupção nas máquinas, nelas foram executados programas que realizavam muitas operações com o disco rígido e sobrecarregavam a rede. Os teclados e mouses também foram exaustivamente utilizados. Os sistemas foram também configurados para executar 1000 interrupções de relógio por segundo, ao invés de 100, que é o padrão.

O *Tx*, antes de enviar a mensagem pela rede, enviava um sinal para a porta paralela de sua máquina que era captado por um ociloscópio. Quando o programa *Rx* recebia a mensagem, também enviava uma mensagem para sua porta serial, permitindo que o ociloscópio registrasse o tempo decorrido entre o envio e a entrega da mensagem (T_{com}).

Antes de iniciar e após finalizar cada fase, os arquivos */proc/interrupts* das máquinas eram consultados, permitindo o cálculo do número médio de interrupções ocorridas nas duas máquinas, durante as medições. Cada fase das medições foi executada no período de 30 *min*, e os resultados são apresentados na Tabela 2.

A Tabela 2 mostra um aumento considerável na taxa de interrupções das máquinas. Neste ambiente de carga, o maior tempo de comunicação registrado foi de 1.7 *ms*. Este valor foi então majorado para 2 *ms* e considerado como valor da variável T_{com}.

Para realização do segundo experimento, foi preparada uma versão especial do serviço com a finalidade de registrar em células de uma matriz, a variação do número de interrupções

		Interrupções					
Ambiente	T_{com}	**Relógio**	**Teclado**	**Mouse**	**eth0**	**disco**	**Total**
Não Carregado	0.5 ms	1800994	1102	1941	1388	88	1806553
Carregado	1.7 ms	1802056	29859	118435	1356391	180703	3668871
Aumento	240%	0%	2610%	6002%	97623%	205200%	103%

Tabela 2: Medição de T_{com} e do total de interrupções

ocorridas na máquina , bem como os tempos T_{prep}, T_{rec} e T_{trat}, durante a execução das fatias de transmissão de *heartbeats*. Esta fatia foi escolhida por ser a que mais processamento executa durante um período TDMA, e portanto, a mais sujeita a atrasos por conta do tratamento de interrupções. As medições dos intervalos de tempo foram feitas com o uso da função *do_gettimeofday* do Linux, cuja precisão é de microsegundos em máquinas da arquitetura PC usadas no experimento. No início e final de cada fase da fatia TDMA, os horários eram medidos e a diferença entre eles era armazenada na matriz de coleta. Como as variáveis T_{prep} e T_{trat} devem considerar o tempo de escalonamento da *thread*, o tempo máximo de escalonamento T_{esc}, que no sistema era de 1 ms foi adicionado ao horário final usado no cálculo do tempo decorrido.

O serviço foi iniciado na máquina com clock de 240Mhz, descrita anteriormente. Em uma primeira etapa, o serviço foi iniciado com o período TDMA configurado para 2000 ms. Após 30 min, estando a máquina sem uso neste período, os valores registrados na matriz foram coletados.

Na segunda etapa, com a máquina carregada como descrito anteriormente, o serviço foi iniciado e deixado para executar por 30 min. Neste período, outras quatro máquinas[7] tiveram também o serviço iniciado, e o máximo de monitoráveis e notificáveis foi inserido nas cinco. Depois, várias entidades monitoráveis tiveram sua execução interrompida, para provocar uma série de notificações pelo serviço, durante a fatia de transmissão de *heartbeats*. No final, os valores registrados na matriz para as variáveis T_{prep}, T_{rec} e T_{trat} foram coletados na primeira máquina.

Os maiores intervalos de tempo registrados para as três variáveis são apresentados na Tabela 3. Os totais de interrupções ocorridos apenas na fatia TDMA de transmissão de

[7] A configuração das 4 outras máquinas não é descrita por não interferir no resultado deste experimento.

heartbeats podem ser vistos na Tabela 4.

	Variáveis		
Ambiente	T_{prep}	T_{rec}	T_{trat}
Não Carregado	3 ms	2 ms	2 ms
Carregado	47 ms	20 ms	43 ms
Aumento	1467%	900%	2050%

Tabela 3: Medição de T_{prep}, T_{rec} e T_{trat}

	Interrupções						
Ambiente	**Relógio**	**Teclado**	**Mouse**	**eth0**	**eth1**	**disco**	Total
Não Carregado	514780	180	4	2135	3720	120	520939
Carregado	514840	16170	93735	981595	3780	123460	1733580
Aumento	0%	8883%	2343200%	45876%	0%	102783%	233%

Tabela 4: Medição do total de interrupções

No ambiente sem carga, o maior valor observado em milhares de períodos TDMA foi de 3 ms para T_{prep}, 2 ms para T_{rec} e 2 ms para T_{trat}. A soma de interrupções ocorridas no período foi de 520939. Já no ambiente carregado, os maiores tempos registrados para as variáveis T_{prep}, T_{rec} e T_{trat} foram respectivamente de 46, 20 e 42 ms, que representam aumentos consideráveis em relação aos tempos registrados no primeiro ambiente. O número total de interrupções foi de 1733580, que equivale a um aumento de 233% em relação ao total do primeiro ambiente.

É importante observar que o método utilizado para medir os valores destas variáveis certamente influenciou os resultados obtidos. As invocações da função *do_gettimeofday*, as operações aritméticas e a manipulação da matriz de resultados consumiram tempo que foram embutidos nos valores medidos. Este é um caso onde o método de observação utilizado modificou o objeto observado. Entretanto, deve-se considerar que, neste caso, esta interferência é positiva, uma vez que aumenta a margem de segurança do cálculo de T_{fatia}, ao majorar os valores das variáveis.

Cálculos Finais

Considerando um ambiente com cinco máquinas, onde ocorrem 1000 interrupções de relógio por segundo (onde $T_{esc} = 1\ ms$) e de posse dos valores das variáveis T_{prep}, T_{com}, T_{rec} e T_{trat}, é possível calcular os valores de T_{fatia} e $T_{periodo}$ usando as fórmulas descritas anteriormente nesta seção. O valor de ρ é constante e igual a 10^{-4}.

Usando as Fórmulas 16 e 17, temos que $T_{sinc} = 34.50345\ ms$ e que $T_{desvio} = 0.0042 \times T_{fatia}$. Logo, usando a Fórmula 8, temos que $\epsilon = 34.50345 + 0.0042 \times T_{fatia}$.

Pela Fórmula 3, temos que $T_{ativ} = 113\ ms$. Logo, usando a Fórmula 2, temos que $T_{fatia} = 2 \times \epsilon + 113$.

Estes cálculos resultam no sistema de equações com duas variáveis composto pelas Fórmulas 18 e 19.

$$\boxed{\epsilon = 34.0034 + 0.0042 \times T_{fatia}} \tag{18}$$

$$\boxed{T_{fatia} = 2 \times \epsilon + 112} \tag{19}$$

Resolvendo o sistema de equações, temos que $\epsilon = 35.27\ ms$ e que $T_{fatia} = 183.55\ ms$. Para aumentar a margem de segurança do cálculo, o valor de ϵ foi majorado para 36 ms, alterando o valor de T_{fatia} para 185 ms.

Usando a Fórmula 1, temos que $T_{periodo} = 1295\ ms$.

Considerando um ambiente também com cinco máquinas, onde ocorrem 100 interrupções de relógio por segundo, o valor de T_{esc} é igual a 10 ms. Isso requer que os valores das variáveis T_{prep} e T_{trat} sejam atualizados. Conforme explicado anteriormente, durante a realização dos experimentos, os valores coletados destas duas variáveis foram acrescidos de 1 ms, para considerarem o maior valor de T_{esc}. Neste novo ambiente, onde ocorrem 10 vezes menos interrupções de relógio, os valores de T_{prep} e T_{trat} devem ser acrescidos de 9 ms para integrarem 10 ms como valor de T_{esc}. Desta forma, foram consideradas $T_{prep} = 56\ ms$ e $T_{trat} = 53\ ms$.

Realizando os mesmos cálculos detalhados anteriormente, temos que T_{fatia} é igual a 211 ms e $T_{periodo}$ é igual a 1477 ms.

Os experimentos realizados para estes cálculos consideraram um ambiente muito menos potente do que é normalmente encontrado hoje em dia na maioria das redes locais. A configuração e o desempenho de máquinas mais modernas permitiriam adotar valores bem menores para $T_{periodo}$, diminuindo assim o tempo de latência mínimo de detecção de falha do serviço.

Em máquinas Pentium IV, com 632 $Mbytes$ de memória RAM, clock de 1.7 GHz, disco rígido de 40 $GBytes$, conectados a uma rede Ethernet de 100 $Mbps$, com a distribuição do Linux Mandrake 9.0, kernel versão 2.4.19, os mesmos experimentos foram repetidos e valores menores foram registrados para as variáveis, conforme pode ser visto nas Tabelas 5 e 6.

	Variáveis			
Ambiente	T_{prep}	T_{com}	T_{rec}	T_{trat}
Não Carregado	1 ms	0.5 ms	2 ms	2 ms
Carregado	31 ms	2 ms	4 ms	34 ms
Aumento	300%	300%	100%	160%

Tabela 5: Medição de T_{prep}, T_{com}, T_{rec} e T_{trat} em máquina mais moderna

	Interrupções						
Ambiente	**Relógio**	**Teclado**	**Mouse**	**eth0**	**eth1**	**disco**	Total
Não Carregado	514810	201	12	5012	3980	130	524145
Carregado	514790	19557	147137	1442738	4009	121500	2249731
Aumento	0%	960%	1226000%	28600%	0%	93361%	332%

Tabela 6: Medição do total de interrupções em máquina mais moderna

Considerando este resultado, os cálculos foram repetidos, considerando um domínio de detecção com 5 máquinas. Em um sistema de 1000 interrupções de relógio por segundo, T_{fatia} é igual a 112 ms e $T_{periodo}$ é igual a 784 ms. Em um sistema de 100 interrupções de relógio por segundo, T_{fatia} é igual a 138 ms e $T_{periodo}$ é igual a 966 ms.

Cuidados para garantir a adequação de $T_{periodo}$

Esta implementação permite que o Delphus seja iniciado com qualquer um dos quatro valores, desde que a taxa de interrupções de relógio e velocidade da máquina sejam compatíveis com o tamanho de período escolhido. Estes testes de compatibilidade são realizados

na iniciação do serviço na máquina, durante a fase de validação dos argumentos da função *start_monitoring*, citado na seção 4.2.2 e detalhado a seguir.

Em primeiro lugar, tanto a taxa de interrupções de relógio quanto a velocidade da máquina são pesquisadas. A velocidade da CPU é medida através da execução da função *calculate_cpu_speed*. Esta função retorna o tempo gasto para realizar um grande volume de cálculos numéricos. Ela é executada em modo *kernel* e por isso não é interrompida pelo escalonador de processos. A taxa de interrupções de relógio é calculada de uma maneira bem simples. O serviço faz o processo dormir por 1 segundo e verifica quantas interrupções de relógio ocorreram no período: se 100 ou 1000. Assim descobre qual das taxas é a utilizada pelo sistema.

Em segundo lugar, é conferido se o período TDMA informado como argumento pelo usuário é compatível com o limite inferior estabelecido para o sistema. A Tabela 7 relaciona os tamanhos de $T_{periodo}$ com as configurações mínimas do sistema exigidas para poderem ser adotados pelo Delphus.

	Configurações Mínimas do Sistema	
$T_{periodo}$	**Interrupções de Relógio**	**CPU**
784 ms	1000/$segundo$	Pentium 1.7 Ghz
966 ms	100/$segundo$	Pentium 1.7 Ghz
1295 ms	1000/$segundo$	Pentium 240 Mhz
1477 ms	100/$segundo$	Pentium 240 Mhz

Tabela 7: Relação entre $T_{periodo}$ e configurações do sistema

O tamanho de período 784 ms é o mais restritivo de todos. Ele se destina a sistemas cuja taxa de interrupções de relógio seja igual a 1000 por segundo e a CPU tenha velocidade de um Pentium com clock de 1.7 Ghz. Já o tamanho de período 1477 ms é o menos restritivo. Ele se destina a sistemas cuja taxa de interrupções de relógio seja igual à padrão de 100 por segundo e a CPU tenha velocidade de um Pentium com clock de apenas 240 Mhz.

Se o valor requerido para $T_{periodo}$ não for compatível com o sistema, um código de erro é retornado para a função *start_monitoring*, que por sua vez interrompe a iniciação do serviço e retorna um código de erro para o processo que a invocou. Se for igual ou superior, prossegue a iniciação. Se este argumento for informado igual a zero, o menor valor de $T_{periodo}$ para a

configuração do sistema será adotado, de acordo com a tabela 7.

Cada novo módulo no domínio de detecção também verifica o tamanho adotado pelo líder, pela frequência de recebimento de mensagens de sincronização e se não for a mesma que pretende adotar, retorna um código de erro. Este tratamento garante que todos os módulos do serviço trabalharão com um mesmo tamanho do período TDMA, compatível com a taxa de interrupções de relógio de todas as máquinas.

4.2.5 Consulta do Estado dos Monitoráveis

A criação dos heartbeats depende da consulta do estado dos monitoráveis pelo módulo do serviço. A cada fatia de *heartbeat* de uma máquina do domínio, a *thread do_period* deve selecionar os processos monitoráveis programados para transferir seu *heartbeat* no período corrente e procurar seu registro na tabela de processos do sistema operacional usando a função interna do Linux *find_task_by_pid*. Se este registro for encontrado, e o estado do processo não for *zombie*, o processo é considerado correto. Caso contrário, seu estado é considerado falho.

4.2.6 Formato das Mensagens

As mensagens do serviço são estruturadas de forma a ocupar no máximo 10 *bytes* (desconsiderando, é claro, o espaço gasto com informações de controle dos protocolos de rede). Essa limitação se deve à interligação existente entre este trabalho e o trabalho de Brito [13]. O dispositivo de comunicação desenvolvido por Brito para implementar o canal de comunicação síncrono de versões futuras do Delphus não possui a mesma velocidade de transmissão de uma rede Ethernet. Por este motivo, foi necessário economizar espaço nas mensagens síncronas, para que o tempo de latência de detecção em versões futuras do serviço não tivesse que ser estendido. O tamanho máximo adotado foi de 10 *bytes* para as mensagens síncronas. As mensagens assíncronas, que trafegam pela rede assíncrona, não tiveram um tamanho limite estabelecido, mas foram estruturadas de forma similar, para ocupar o menor espaço possível, minimizando seu impacto no desempenho do sistema.

Todas as mensagens do serviço armazenam no primeiro byte um código chamado

método da mensagem, que a identifica exclusivamente no seu canal. As *threads sync_rec* e *async_rec* consultam este byte para identificar qual rotina de tratamento de mensagem deverá ser invocada. A seguir, é mostrado o formato de cada mensagem síncrona.

Sincronização

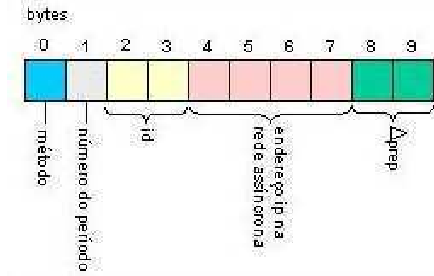


Figura 23: Formato da Mensagem de Sincronização

Conforme mostrado pela Figura 23, a mensagem de sincronização utiliza o byte de índice 1 para armazenar o número do período TDMA vigente. A seguir, nos bytes de ordem 2 e 3, o identificador no domínio da máquina líder é armazenado. Os bytes de ordem 4 a 7 são usados para armazenar os octetos do endereço IP da rede assíncrona da máquina líder. Os últimos bytes, de ordem 8 e 9, são utilizados para armazenar o valor Δ_{prep}, que é usado na sincronização dos períodos.

Proposta de Liderança

Conforme mostrado pela Figura 24, a mensagem de proposta de liderança utiliza os bytes de índice 1 e 2 para armazenar o identificador no domínio da máquina que tenta ser líder do domínio. Os bytes de ordem 3 a 6 são usados para armazenar os octetos do endereço IP da máquina na rede assíncrona.

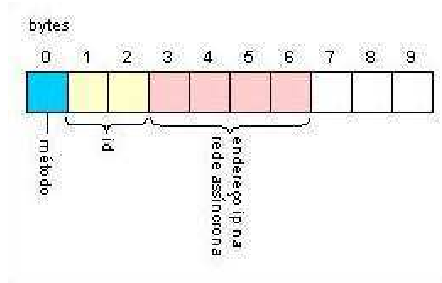


Figura 24: Formato da Mensagem de Proposta de Liderança

Anúncio de Admissão de Monitorável

A Figura 25, mostra a estrutura de uma mensagem de anúncio de admissão de uma nova máquina no domínio de detecção. Os bytes de índice 1 e 2 são utilizados para armazenar o identificador da máquina admitida no domínio. Os bytes de ordem 3 a 6 são usados para armazenar os octetos do endereço IP da máquina na rede assíncrona.

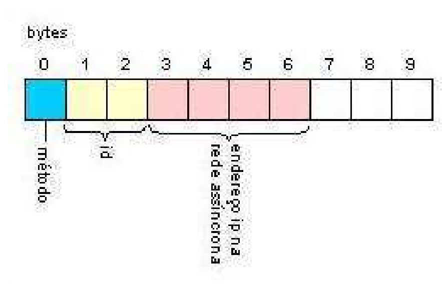


Figura 25: Formato da Mensagem de Anúncio de Admissão de Máquina Monitorável

A Figura 26, mostra a estrutura de uma mensagem de anúncio de admissão de um processo monitorável no domínio de detecção. Esta é a maior mensagem síncrona que existe atualmente. Os bytes de índice 1 e 2 são utilizados para armazenar o identificador da máquina onde executa o processo. Os bytes de ordem 3 e 4 armazenam o identificador do processo no domínio. O *pid* do processo é armazenado nos bytes de ordem 5 e 6. O byte 7 armazena o número de ordem do primeiro período em que o processo monitorável deve ter seus *heartbeats* transmitidos em um super período[8]. Este número de ordem é tomado como

[8] O conceito de super período foi explicado na Seção 3.3.3.

base pelos módulos do serviço para calcular os períodos em que o monitorável deverá ter seus *heartbeats* transmitidos. Este não necessariamente equivale ao período atual, podendo ser um já passado.

No byte de ordem 8 é armazenado o valor que indica de quantos em quantos períodos TDMA deverá haver uma transmissão de *heartbeat* do processo. Por fim, o byte de ordem 9 indica quantos períodos faltam para que o monitorável comece a ter seus heartbeats transmitidos. Este byte foi usado para esta finalidade a fim de facilitar o processo de decodificação da mensagem nos módulos e diminuir o processamento necessário para adicionar o processo na lista dos monitoráveis. Em versões futuras do serviço, quando o tamanho das mensagens síncronas for aumentado e houver pelo menos mais um byte livre nesta mensagem, este valor poderia ser calculado a partir dos armazenados nos bytes anteriores. Desta forma, os 2 bytes livres poderiam ser usados para aumentar o tamanho dos identificadores da máquina e do processo no domínio.

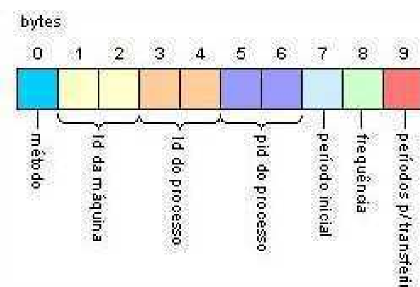


Figura 26: Formato da Mensagem de Anúncio de Admissão de Processo Monitorável

Anúncio de Remoção de Monitorável

Conforme mostrado pela Figura 27, a mensagem de anúncio de remoção de monitorável utiliza os bytes de índice 1 e 2 para armazenar o identificador no domínio da máquina a ser removida do domínio.

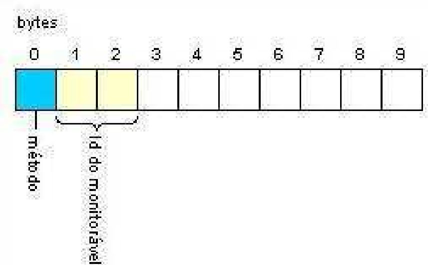


Figura 27: Formato da Mensagem de Anúncio de Remoção de Monitorável

Heartbeat

As mensagens de *heartbeat* armazenam nos bits dos bytes de ordem 1 a 9 o estado dos processos que devem transmitir no período corrente. O valor 1 indica que estão corretos. O valor 0 indica que estão falhos.

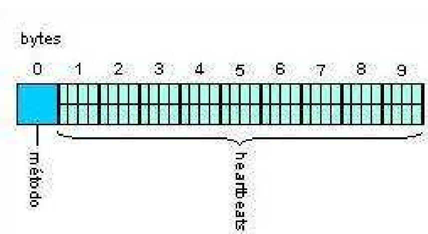


Figura 28: Formato da Mensagem de Heartbeat

A seguir, é mostrado os formatos das mensagens assíncronas.

Requisição de Admissão

Conforme mostrado pela Figura 29, a mensagem de requisição de admissão de uma máquina no domínio utiliza os bytes de ordem 1 e 2 para armazenar o identificador no domínio pretendido para a máquina. Os bytes de ordem 3 a 6 são usados para armazenar os octetos do endereço IP da máquina na rede síncrona.

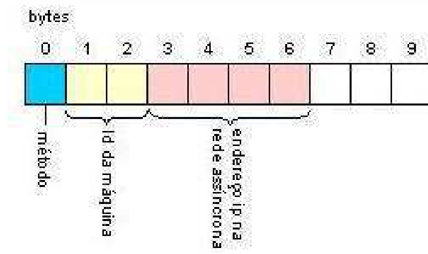


Figura 29: Formato da Mensagem de Requisição de Admissão de Máquina Monitorável

A Figura 30 mostra a estrutura de uma mensagem de requisição de admissão de um processo monitorável no domínio. Os bytes de ordem 1 e 2 são usados para armazenar o identificador da máquina onde reside o processo no domínio. Os bytes de ordem 3 e 4 armazenam o identificador do processo no domínio. Os bytes de ordem 5 e 6 armazenam o *pid* do processo. Os bytes de ordem 7 a 9 armazenam informações sobre a frequência de transmissão de *heartbeats*, descritas anteriormente para a mensagem de anúncio de admissão de processo monitorável. Os dois últimos bytes indicam em que período a requisição foi feita. Apenas requisições de admissão do período corrente são agendadas para anúncio no próximo período. Isto garante que os cálculos sobre a transmissão de heartbeats dos monitoráveis estarão atualizados, quando ele for adicionado às listas de monitoráveis dos módulos do domínio.

Requisição de Remoção de Monitorável

Conforme mostrado pela Figura 31, a mensagem de proposta de liderança utiliza os bytes

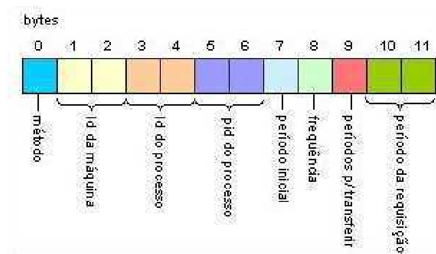


Figura 30: Formato da Mensagem de Requisição de Admissão de Processo Monitorável

de índice 1 e 2 para armazenar o identificador no domínio da máquina a ser removida do domínio.

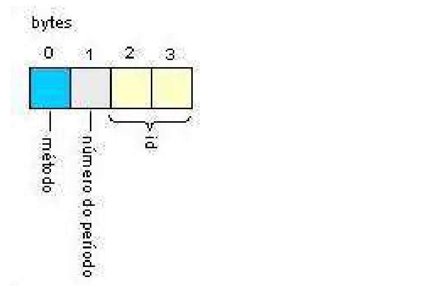


Figura 31: Formato da Mensagem de Requisição de Remoção de Monitorável

Recusa de Admissão de Monitorável

Uma mensagem de recusa apenas armazena um código do erro que motivou a recusa do ingresso de um monitorável no domínio de detecção. Este código ocupa os bytes de ordem 1 e 2, conforme mostrado pela Figura 32.

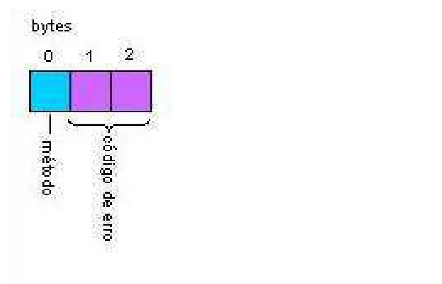


Figura 32: Formato da Mensagem de Recusa de Admissão de Monitorável

Atualização

A mensagem de atualização é a maior utilizada pelo serviço. Ela possui os dados de todas as entidades monitoráveis já cadastradas no domínio. Conforme mostrado na Figura 33, ela pode ser entendida como a junção de várias mensagens do tipo Anúncio de Admissão. As mensagens de anúncio das máquinas do domínio precedem as mensagens de admissão dos processos monitoráveis que nelas executam. Cada bloco de mensagens de anúncio referentes a uma máquina é intercalado por um caracter especial de finalização de bloco. Outro caracter

especial finaliza todo o conjunto de monitoráveis.

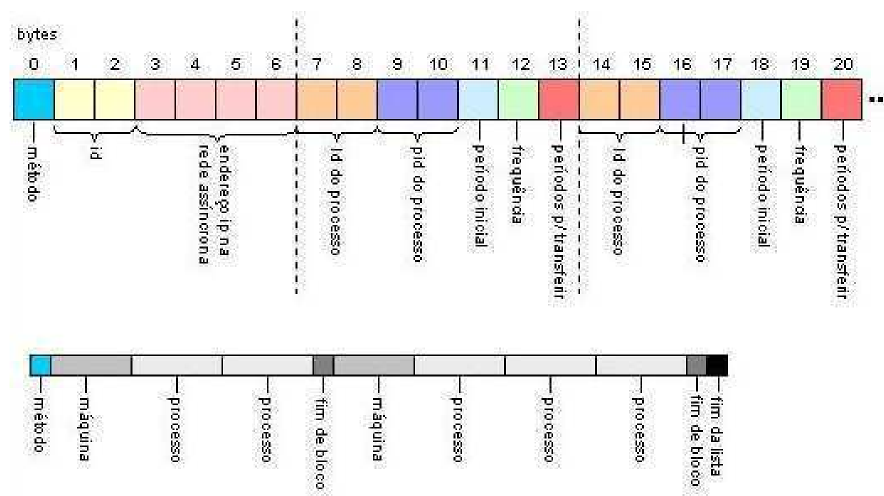


Figura 33: Formato da Mensagem de Atualização da Lista de Monitoráveis

Compromisso

Como mostrado na Figura 34, uma mensagem de Compromisso armazena nos bytes de ordem 1 e 2 o identificador da máquina pronta para fazer parte do domínio de detecção. Os bytes de ordem 3 a 6 armazenam os octetos do endereço IP da máquina na rede assíncrona.

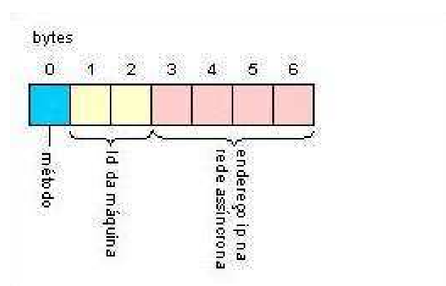


Figura 34: Formato da Mensagem de Compromisso

4.2.7 Notificação de Eventos

A notificação de eventos de monitoria pode ser feita nesta implementação de duas formas: através do envio de sinais a processos notificáveis, ou através da invocação de métodos de objetos notificáveis. A forma adotada dependerá do tipo da entidade notificável.

A notificação de processos notificáveis é feita da seguinte forma. Em primeiro lugar, durante o cadastro de um processo notificável, é informado para cada evento de monitoria (admissão de monitorável, remoção de monitorável e falha em monitorável), o sinal que deve ser enviado para o processo, assim que o evento ocorrer. Desta forma, quando ocorre um evento no serviço (como por exemplo a admissão de um novo monitorável, durante o tratamento de uma mensagem de anúncio), a função *notify* é invocada. Ela percorre toda a lista de processos notificáveis e seleciona todos os que estão interessados naquele evento com aquele monitorável específico, ou com qualquer monitorável. Uma vez que esta seleção tenha sido feita, cada notificável tem seu *pid* e o sinal cadastrado para aquele evento consultado. Esses dados são usados na invocação da função do *kernel kill_proc_info*, que envia o sinal para o processo. Esta função é invocada nas funções de tratamento das mensagens de anúncio de admissão de monitorável, anúncio de remoção de monitorável e de *heartbeat*.

A notificação de objetos é diferente, mas baseia-se também no uso de sinais. Como será explicado na Seção 4.3.2, o objeto da API Java responsável pelo cadastro de entidades notificáveis é o *Notifier*. Os objetos notificáveis são armazenados em uma lista interna do *Notifier*. Essas entidades são desconhecidas do módulo de núcleo do Delphus. A forma adotada pelo *Notifier* para se tornar ciente da ocorrência de eventos e assim ser capaz de notificar os objetos notificáveis é cadastrar o processo corrente como um processo notificável, como qualquer outro. Para isso, ele utiliza como identificador uma combinação exclusiva entre caracteres especiais como # e !.

O sinal usado para esta notificação é o SIGWINCH, para o qual os processos no Linux têm o comportamento padrão de ignorar ao receber. Este sinal é cadastrado para ser recebido em caso de ocorrência de qualquer um dos eventos de monitoria.

Usando a API JavaSignals [35], que permite objetos Java tratar interrupções do Linux, o *Notifier* implementa uma rotina para tratar o sinal SIGWINCH. Ao recebê-lo, o *Notifier* consulta a lista das entidades monitoráveis atualmente cadastradas e seus estados. O passo seguinte é identificar os eventos ocorridos desde o último sinal recebido. Por fim, o *Notifier* consulta sua lista de objetos notificáveis e invoca os métodos de notificação (explicados na Seção 4.3.2) daqueles interessados nos eventos recém ocorridos.

Convém informar que o sinal SIGWINCH só é configurado para esta finalidade, caso

objetos notificáveis sejam cadastrados na API Java. Se nenhum objeto for cadastrado como notificável, esta configuração não será realizada.

4.2.8 Condições de Corrida

Em um ambiente multitarefa como o Linux, vários processos podem utilizar o Delphus concorrentemente e gerar condições de corrida, que se não forem devidamente tratadas, podem provocar inconsistências. Imagine a simples situação onde dois processos tentam concorrentemente se cadastrar como notificáveis. Os processos notificáveis são armazenados em uma pilha, ou seja, uma lista simplesmente encadeada FILO (*first in last out*). Cada novo elemento da lista é inserido no topo da lista. Seu ponteiro *next* é ajustado para apontar para o topo atual e o ponteiro do topo (ponteiro de entrada na lista) é atualizado para apontar para o novo elemento. Durante a manipulação da lista de monitoráveis, no cadastro do primeiro processo, antes que este ponteiro seja alterado para apontar para o elemento recém incluído, o escalonador de processos do sistema operacional poderia conceder ao segundo processo a chance de usar a CPU. Este segundo processo encontraria a lista em um estado inconsistente e tentaria inserir nela um nó com suas informaçoes, sem considerar a existência do primeiro elemento. Supondo que o segundo processo conseguisse concluir o cadastro, quando o primeiro voltasse a utilizar a CPU, concluiria seu trabalho atualizando o ponteiro do topo da lista para apontar para seu elemento. Isso provocaria a perda do elemento do segundo processo. Situações como esta impediriam que o serviço funcionasse corretamente.

Entretanto, alguns fatores impedem que elas ocorram. Em sistemas com apenas um processador, não ocorre preempção de um processo em *kernel mode*. Isso quer dizer que, a menos que ele se torne bloqueado, ele não é trocado na CPU por outro processo. Isso garante atomicidade das tarefas que o processo executar neste modo. Cada função da API do serviço, em última instância, invoca uma *system call* para se comunicar com o módulo do serviço e durante a execução de uma *system call*, o processo passa a estar em *kernel mode*. Como as funcionalidades disponíveis na API do serviço são executadas com apenas uma invocação de *system call*, não há risco de inconsistências causadas pela comutação da CPU pelos processos. Interrupções podem suspender a execução de um processo em *kernel mode*,

mas quando sua rotina de manipulação termina de executar, o mesmo processo volta a usar a CPU.

Uma outra possível situação de condição de corrida seria o acesso concorrente às estruturas de dados do serviço por funções do próprio serviço, que executam em *kernel mode*. Por exemplo, enquanto a lista de processos monitoráveis estivesse sendo atualizada, e em um estado inconsistente, a *thread do_period* poderia precisar consultá-la para enviar uma mensagem de *heartbeats*. Essa situação não ocorre no Delphus porque as tarefas são sincronizadas para ocorrerem em espaços de tempo diferentes (nas fatias TDMA), de modo que a cada momento, apenas um tipo de operação nas estruturas é realizada. Além do mais, a maneira com que os elementos são inseridos nas listas encadeadas permite que as consultas apenas desconsiderem elementos com inserção em andamento, tornando possível a consulta aos elementos previamente existentes.

Em sistemas multiprocessados, onde existem mais de uma CPU, mais de um processo podem estar em *kernel mode*. Isso exige que outras técnicas sejam empregadas para garantir atomicidade das operações sujeitas a condições de corrida. O Linux dispõe de vários recursos para tratar estas situações, como desabilitação de interrupções, uso de semáforos, barreiras de memória, bloqueio de acesso a estruturas, dentre outros [12]. A sincronização das atividades do serviço, explicada anteriormente, garante a não ocorrência de muitas condições de corrida. As únicas situações em que ela poderia ocorrer seriam em operações que não dependem de uma fatia do período TDMA, como o cadastro de processos notificáveis. Dois processos poderiam estar em *kernel mode* executando em cada CPU a mesma função de cadastro de notificável, descrita no início desta seção. Para evitar tais condição de corrida, *spin locks* foram usados nas estruturas de dados do serviço [12]. Através das funções *spin_lock*, *spin_unlock* e *spin_unlock_wait*, cada região crítica do código do serviço foi protegida para apenas um processo ter acesso. Vale lembrar que em ambientes monoprocessados, essas funções não realizam qualquer computação.

4.3 API

A API do Delphus foi implementada tanto em C quanto em Java. Sendo Java uma linguagem de mais alto nível que o C e orientada a objetos, sua versão da API tornou-se muito mais sofisticada que a implementada em C. Entretanto, as duas versões da API foram construídas de forma integrada. A API Java apoia-se sobre a tecnologia JNI (*Java Native Interface*) para ter acesso à biblioteca de funções da API em C. Essa integração evita redundância de código entre as versões da API, facilita os testes do serviço, homogeniza as funcionalidades das duas interfaces e consequentemente, promove a confiabilidade e facilidade de uso do serviço. A desvantagem da adoção desta estrutura, é que a API Java deixa de ser beneficiada pelos mecanismos de otimização de execução de código empregados pela máquina virtual Java, como a compilação *Just in Time* (JIT). Entretanto, considerando a simplicidade das funções da API C e a baixa frequência com que, normalmente, uma aplicação cliente invoca funções da API, esta estrutura não causa perdas consideráveis no desempenho das aplicações.

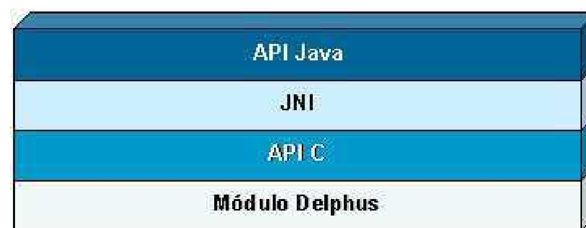


Figura 35: Estrutura das APIs

4.3.1 API C

Todas as funções da API C retornam um valor inteiro, que indica o resultado da execução. Quando positivo, indica que a função foi executada com sucesso. Quando negativo, indica a ocorrência de um erro. Os valores positivos dependem da lógica da função (sendo 1 por *default*) enquanto os valores negativos seguem a classificação de erros apresentada na Tabela 8.

A seguir, a descrição das situações onde cada um desses erros ocorrem.

Código	Constante	Descrição
-1	UNLOADED_SERVICE_ERROR	*O serviço não foi carregado ainda*
-2	LOADED_SERVICE_ERROR	*O serviço já foi carregado*
-3	AUTHORIZATION_ERROR	*Usuário não autorizado a executar função*
-4	LIMIT_REACHED_ERROR	*Não é possível inserir mais um ítem*
-5	ELEMENT_NOT_FOUND_ERROR	*Elemento não pôde ser encontrado*
-6	VALIDATION_ERROR	*Erro de validação de valor*
-7	REQUIRED_VALUE_ERROR	*Valor requerido não informado*
-8	VALUE_NOT_EXCLUSIVE_ERROR	*Valor não exclusivo*
-9	ABNORMAL_ERROR	*Condição anormal do serviço*
-10	MEMORY_ALOCATION_ERROR	*Erro de alocação de Memória*
-11	NETWORK_ERROR	*Erro de rede*

Tabela 8: Códigos de Erro da API C

UNLOADED_SERVICE_ERROR

O erro UNLOADED_SERVICE_ERROR ocorre quando uma função necessita que o serviço esteja iniciado na máquina onde foi invocada (como a que retorna a lista de monitoráveis corretos, por exemplo) e o serviço encontra-se inativo.

LOADED_SERVICE_ERROR

O erro LOADED_SERVICE_ERROR ocorre quando uma função necessita que o serviço não esteja iniciado na máquina onde foi invocada (como a *stop_monitoring*, por exemplo) e o serviço já se encontra ativo.

AUTHORIZATION_ERROR

O erro AUTHORIZATION_ERROR ocorre quando um usuário invoca uma função cujo uso é reservado ao super usuário (*start_monitoring* e *stop_monitoring*), ou quando tenta cadastrar processos que não são seus como entidades monitoráveis ou notificáveis. Apenas o super usuário tem permissão de invocar todas as funções, e de cadastrar qualquer processo como entidade do Delphus.

VALIDATION_ERROR

O erro VALIDATION_ERROR ocorre quando um argumento de uma função é informado em um formato incorreto, ou extrapolando seus limites superior ou inferior. Um endereço IP igual a "167.88" ou um valor negativo para *pid* são alguns exemplos.

REQUIRED_VALUE_ERROR

O erro REQUIRED_VALUE_ERROR ocorre quando um argumento obrigatório é informado igual a nulo durante a invocação de uma função. Um exemplo disso é invocar a função *add_monitorable*, informando seu argumento *id* como nulo.

VALUE_NOT_EXCLUSIVE_ERROR

O erro VALUE_NOT_EXCLUSIVE_ERROR ocorre quando se tenta usar um valor não exclusivo para um atributo definido como exclusivo. Por exemplo, ao tentar cadastrar um processo monitorável invocando a função *add_monitorable*, usar como argumento *id* um identificador já usado por outro monitorável previamente cadastrado.

LIMIT_REACHED_ERROR

O erro LIMIT_REACHED_ERROR ocorre quando se pretende inserir um elemento em uma lista cuja capacidade está esgotada. Um exemplo deste erro ocorre quando, em um domínio de detecção limitado a um número máximo de máquinas igual a n, tenta-se inserir $(n+1)$ máquinas.

ELEMENT_NOT_FOUND_ERROR

Quando uma função busca por um elemento em uma lista e não o encontra, o erro ELEMENT_NOT_FOUND_ERROR é retornado. Isso ocorre, por exemplo, quando se busca uma entidade monitorável por um identificador, usando a função *get_monitorable* e esta entidade não é encontrada.

ABNORMAL_ERROR

O erro ABNORMAL_ERROR ocorre em situações raras e nocivas ao funcionamento do

serviço. Um exemplo seria a ocorrência de um erro de alocação de memória no momento que o serviço tentasse criar uma mensagem de *heartbeats*. Isso impediria que a mensagem fosse enviada na fatia TDMA adequada, provocando falsas suspeições dos seus monitoráveis, por parte dos módulos do serviço. A ocorrência desse erro provoca uma falha por parada da máquina. Uma *thread* com prioridade de tempo real é criada para consumir todo o tempo de CPU da máquina e provocar sua falha. Desta forma, as demais máquinas a considerarão falha corretamente.

A seguir, são apresentadas as principais funções da API C, mostrando suas assinaturas, explicando seu comportamento, e descrevendo seus argumentos e valores de retorno. As funções são divididas pelos mesmos grupos de funcionalidades definidos no projeto da API, vistos anteriormente na Seção 3.4.

Administração do Serviço

- **int start_monitoring(char *id, char *params)** Esta função é utilizada para inscrever uma máquina em um domínio de detecção. Ela recebe no argumento *id*, o identificador proposto para a máquina no domínio de detecção e no argumento *params*, um vetor com informações secundárias para iniciação do serviço. Na versão atual do Delphus, este vetor deve conter os endereços IP dos canais síncrono e assíncrono e o tempo em milisegundos pretendido para um período TDMA. Futuras versões do serviço que necessitarem de argumentos diferentes, deverão fornecer neste argumento, um vetor com outros elementos.

  O tamanho informado para um período TDMA deve estar entre os valores calculados para a variável $T_{periodo}$ na Seção 4.2.4: 784 ms, 966 ms, 1295 ms ou 1477 ms. A partir deste valor, o serviço identifica os valores que devem ser adotados para as demais variáveis ϵ, T_{fatia}, T_{prep}, T_{com}, T_{rec} e T_{trat}, que são os mesmos usados nos cálculos da Seção 4.2.4.

- **int stop_monitoring** Remove a máquina, onde o processo invocador está executando, do domínio de detecção.

- **int is_started(char *id)** Esta função informa se uma máquina está ou não correntemente sendo parte de um domínio de detecção do serviço. Ela retorna o valor 1 caso o serviço esteja executando naquela máquina e 0, caso contrário.

Configuração de Monitoração

- **int add_monitorable(char *id, int pid, unsigned long int detection_time)** Esta função é utilizada para adicionar um processo novo para ser monitorado pelo serviço. Ela possui os seguintes argumentos: *id*, o identificador no serviço, para o processo; *pid*, o identificador do processo da entidade monitorável (se igual a 0, o *pid* do processo invocador será utilizado); e *detection_time*, o intervalo máximo de tempo, com precisão de milisegundos, dentro do qual uma falha no processo deve ser detectada e conhecida por todos os módulos do serviço.

  O último parâmetro da função anterior define a qualidade do serviço requisitada. É importante notar que a latência máxima de detecção é garantida apenas com relação ao tempo que o módulo de detecção de falha local detecta a falha da entidade monitorável e envia os sinais apropriados para a entidade notificável correspondente. Nenhuma garantia é dada para o tempo que a entidade notificável vai efetivamente tratar o sinal. Por outro lado, como será explicado mais tarde, qualquer processo pode consultar seu módulo de detecção de falhas local para saber se uma entidade monitorável está correta. Quando o serviço retorna a informação que uma entidade está correta no tempo T, é garantido que a entidade estava correta no tempo $T - detection_time$.

- **int remove_monitorable(char *id)** Remove um processo da lista corrente de monitoráveis. Recebe como argumento apenas a identificação no serviço do processo a ser removido.

- **int get_monitorable(char *machine_id, struct monitorable **list)** Esta função preenche o argumento *list* com uma lista de entidades monitoráveis, e retorna a quantidade de entidades selecionadas. Se o argumento *machine_id* for informado, a lista será restrita aos processos monitoráveis da máquina no domínio de detecção identi-

ficada pelo argumento. Se o argumento for igual a nulo, todas as entidades monitoráveis serão retornadas na lista.

- **int find_monitorable(char *id, struct monitorable **monitorable)** Esta função retorna no argumento *monitorable* a entidade monitorável identificada pelo argumento *id*, que é a identificação do monitorável no domínio de detecção.

Notificação de Falha

- **int is_correct(char *id, struct timeval *time)** Esta função informa se um monitorável está correto ou não. Ela recebe como argumento o identificador no serviço da entidade monitorável cujo estado está sendo consultado. Retorna o valor 1 se o monitorável estiver correto, e 0 caso contrário. O argumento *time* é utilizado para armazenar o tempo local no qual a informação foi coletada.

- **int is_faulty(char *id, struct timeval *time)** Esta função é o complemento da função anterior. Ela informa se um monitorável falhou ou não. Ela recebe um identificador como parâmetro, e retorna o valor 1 se esta entidade monitorável tiver falhado, e 0 caso contrário. Novamente, o argumento *time* é utilizado para armazenar o tempo local no qual a informação foi coletada.

- **int get_correct(struct monitorable **correct)** Esta função retorna a lista de todas as entidades monitoráveis corretas no argumento *correct*.

- **int get_faulty(struct monitorable **faulty)** Esta função retorna a lista de todas as entidades monitoráveis falhas no argumento *faulty*.

- **int add_monitoring_notifiable(char *id, int pid, char *monit_id, struct event *events)** Esta função configura um processo para ser notificado da ocorrência de eventos no serviço. Seus argumentos são: *id*, a identificação no serviço, da entidade notificável; *pid*, o identificador do processo da entidade notificável (se igual a 0, o *pid* do processo que fez a invocação será utilizado); *monit_id*, o identificador da entidade

monitorável a que os eventos se referem, (se for nulo, o evento se refere a todos os monitoráveis); e *events*, um vetor de estruturas do tipo event, que relaciona os eventos a sinais do sistema operacional que devem ser enviados para a entidade notificável, no caso da ocorrência do evento.

A estrutura *event* possui os seguintes atributos: *code*, o código de identificação do evento e *signal*, o sinal que deve ser enviado ao notificável.

- **int remove_notifiable(char *id)** Remove uma entidade notificável específica, identificada pelo argumento *id*.

- **int get_monitoring_notifiable(char *monit_id, struct notifiable **list)** Esta função preenche o argumento *list* com a lista dos notificáveis de monitoramento, interessados em eventos de monitoria com o monitorável identificado pelo argumento *monit_id*. Se este argumento for informado nulo, todos os processos notificáveis de monitoramento serão retornados. Esta função retorna a quantidade de entidades selecionadas.

- **int find_monitoring_notifiable(char *id, struct notifiable **notifiable)** Esta função retorna no argumento *notifiable* o processo notificável identificado pelo argumento *id*.

4.3.2 API Java

A API Java fornece uma visão orientada a objetos do serviço. Para isso, vários objetos foram definidos para representar as entidades envolvidas com a detecção de falhas. Eles permitem que as aplicações clientes não abandonem o paradigma da orientação a objetos para usar o Delphus, seguindo a proposta apresentada por Felber *et al.* [27] de que o serviço, mesmo sendo do nível do sistema operacional, deveria ser acessível através de “objetos de primeira classe”, ou seja, abstrações do mesmo nível utilizado no resto da aplicação.

Uma implementação da API do serviço deve ser bem estruturada, familiar e amigável aos desenvolvedores, além de favorecer o baixo acoplamento com as aplicações clientes. Para alcançar estes objetivos, padrões de projeto (*design patterns*) foram adotados no projeto da

API Java. Padrões de projeto são soluções consagradas para problemas comuns de projeto de software, já adotadas com sucesso em inúmeros outros projetos pela comunidade de desenvolvedores [31].

Para diminuir o nível de acoplamento das aplicações com o serviço, outros cuidados também foram tomados. Não apenas classes foram criadas para os objetos, mas também interfaces. O objetivo é permitir que as aplicações clientes usem referências aos objetos do tipo das interfaces e não das classes. Isso permite que futuramente, as classes sofram alterações como mudança de nomes ou mudança de hierarquia, sem que seja necessário alterar as aplicações clientes. Desde que as interfaces permaneçam inalteradas, as reestruturações na API serão transparentes para as aplicações. O uso das interfaces permite ainda que as aplicações possam adotar outros serviços de detecção, que implementem as mesmas interfaces que o Delphus, aumentando sua portabilidade.

A Figura 36 mostra o diagrama de classes do pacote *br.edu.ufcg.dsc.delphus.monitorable*. A classe abstrata *MonitorableFactory*, através dos métodos *createMonitorableProcess* e *createMonitorableMachine*, instancia objetos que implementam respectivamente as interfaces *MonitorableProcess* e *MonitorableMachine*. *MonitorableProcess* representa os processos monitorados pelo serviço e *MonitorableMachine* as máquinas que fazem parte do domínio de detecção. Estas interfaces são especializações da interface *Monitorable*, que representa todas as entidades monitoráveis do domínio de detecção.

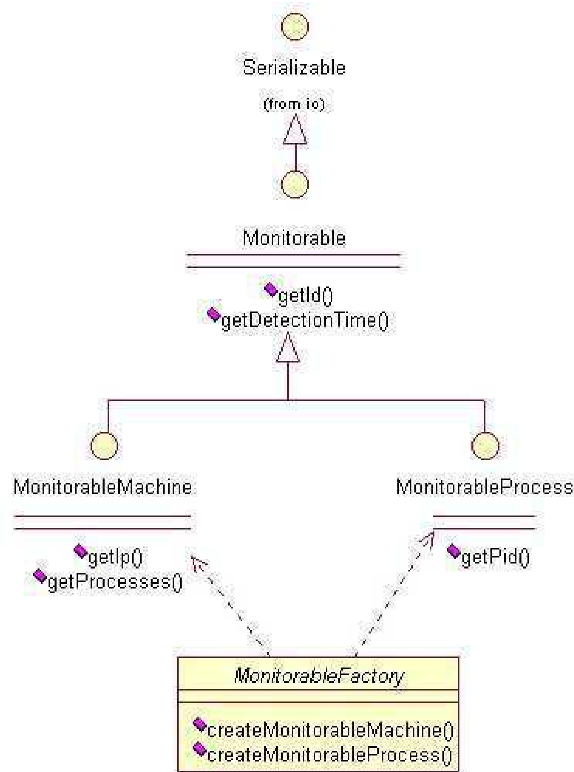


Figura 36: Objetos Monitoráveis

A Figura 37 mostra o diagrama de classes do pacote *br.edu.ufcg.dsc.delphus.notifiable*. A classe abstrata *NotifiableFactory*, através do método *createMonitoringNotifiableProcess*, instancia objetos que implementam a interface *MonitoringNotifiableProcess*. Esta interface representa processos notificáveis do domínio de detecção. Ela é uma especialização da interface *MonitoringNotifiable*, que representa todas as entidades interessadas na notificação de eventos de monitoramento. Outra especialização de *MonitoringNotifiable* é *MonitoringNotifiableObject*, que deve ser implementada por todo objeto que tiver que ser notificado, seguindo o padrão de projeto *Observer* [32]. A interface *MonitoringNotifiable* é uma especialização da interface *Notifiable*, que representa todas as entidades notificáveis, interessadas em qualquer categoria de eventos. Futuramente, outras especializações desta interface poderão existir, para tratar outras categorias de eventos.

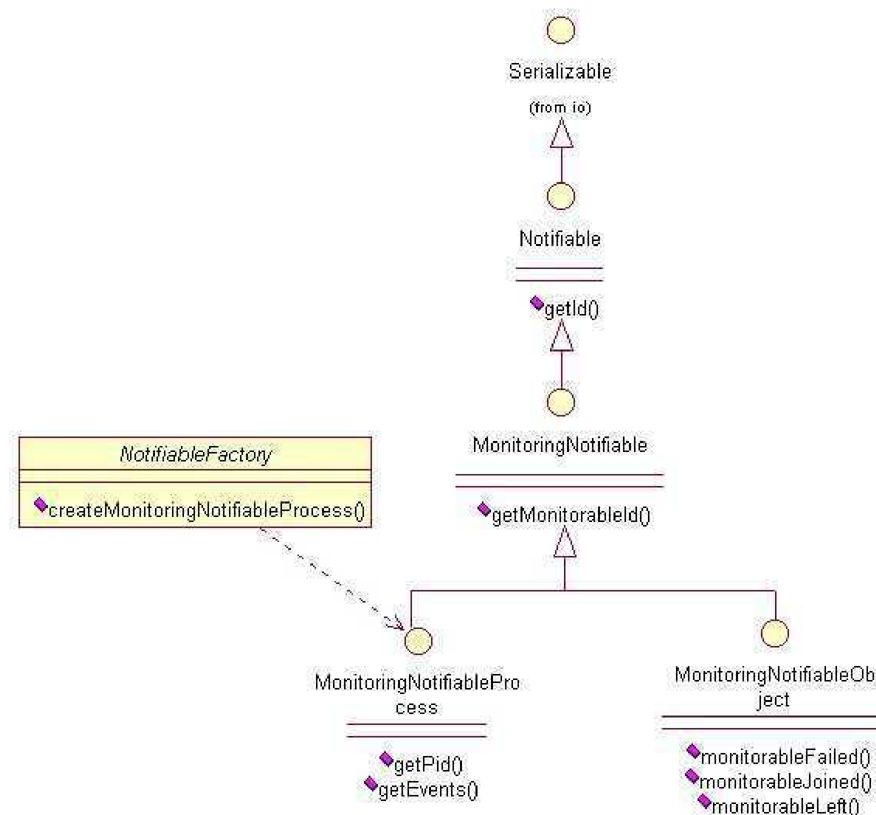


Figura 37: Objetos Notificáveis

As interfaces *Monitorable* e *Notifiable* foram definidas como especializações da interface *Serializable* do pacote *java.io*. Isto permite que seus objetos sejam serializados para transferências entre máquinas da rede, em um ambiente de objetos distribuídos, e que sejam persistidos em arquivos ou bancos de dados.

Para tornar o uso das interfaces obrigatório pelas aplicações, as classes dos objetos foram definidas como classes privadas de classes *AbstractFactory* [33]. Essas classes possuem métodos de criação dos objetos e retornam instâncias das classes privadas, usando suas respectivas interfaces como tipo de retorno. Isto limita às interfaces, os tipos que podem ser usados pelas aplicações para criar referências.

Por exemplo, os objetos processos monitoráveis são criados apenas pela *AbstractFactory MonitorableFactory*, através do método *createMonitorableProcess*. Este método retorna uma instância da classe *MonitorableProcessImpl*, definida como uma classe *friendly*, que imple-

menta a interface *MonitorableProcess*. Classes *friendly* são aquelas que podem ser criadas apenas por objetos de classes do mesmo pacote. Não há como uma aplicação criar diretamente uma instância da classe *MonitorableProcessImpl*, já que seu acesso é restrito às classes do pacote *br.edu.ufcg.dsc.delphus.monitorable*. Portanto, a única forma de criá-la é através da classe *MonitorableFactory*.

Os objetos principais do serviço são: *Manager*, *Scheduler* e *Notifier*. Eles são *Sigletons* [34] criados pela *AbstractFactory ServiceFactory*. A Figura 38 mostra um diagrama de classes do pacote *br.edu.ufcg.dsc.delphus*, onde eles são definidos. Assim como as demais classes *AbstractFactory* vistas anteriormente, *ServiceFactory* retorna instâncias de classes privadas, que implementam interfaces públicas. Estas interfaces são *Notifier*, *Manager* e *Scheduler*.

A maioria dos seus métodos têm relação direta com funções vistas na API C e é através deles que as aplicações têm acesso às funcionalidades do serviço. A seguir, estes objetos são apresentados em detalhes.

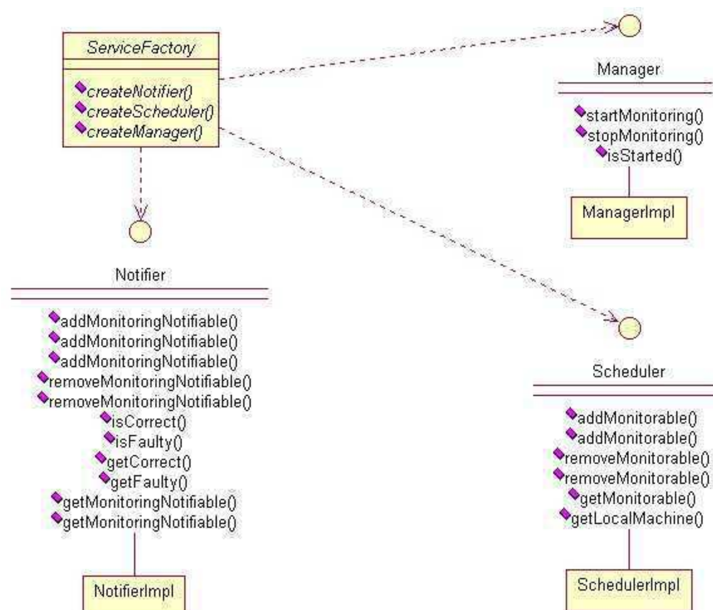


Figura 38: Objetos principais do serviço

Manager

Permite que o serviço seja administrado. Seus métodos repassam as invocações para as funções do grupo Administração do Serviço da API C. A seguir, a descrição de seus métodos.

- **void startMonitoring(String id, List args)** Este método equivale à função *start_monitoring* da API C. O argumento args deve ser preenchido por objetos que representem os endereços IP da rede síncrona e assíncrona e o período TDMA. Os endereços IP devem ser informados como objetos *String* ou *InetAddress* e o período TDMA como *Integer*, ou *String*.

- **void stopMonitoring()** Equivalente à função *stop_monitoring* da API C.

- **boolean isStarted()** Equivalente à função *is_started* da API C.

Scheduler

Permite que processos sejam configurados para serem monitorados pelo serviço. Seus métodos repassam as invocações para as funções do grupo Configuração de Monitoramento da API C. A seguir, a descrição de seus métodos.

- **void addMonitorable(MonitorableProcess monitorableProcess)** Este método equivale à função add_monitorable da API C. Seu único argumento, *monitorableProcess*, contém todos os dados necessários para adicionar um processo monitorável no serviço.

- **void addMonitorable(String id, java.util.Date detectionTime)** Esta versão sobrecarregada do método anterior deve ser utilizada quando o processo atual quiser se cadastrar para ser monitorado. O *pid* do processo corrente será utilizado para identificar o processo monitorável. O Delphus não trabalha com monitoramento de objetos, entretanto, este método permite que objetos usem esta funcionalidade como se fosse para cadastrar a si próprios, abstraindo a identificação do processo onde residem.

- **void removeMonitorable(String id)** Este método equivale à função *remove_monitorable* da API C.

- **void removeMonitorable(Monitorable monitorable)** Esta versão sobrecarregada do método anterior permite que o objeto a ser removido seja identificado por um objeto *Monitorable*, ao invés de diretamente por seu identificador.

- **Collection getMonitorable()** Este método equivale à função *get_monitorable* da API C, invocada com o argumento *machine_id* nulo. Uma coleção de todos os objetos *Monitorable* é retornada.

- **Collection getMonitorable(MonitorableMachine machine)** Este método equivale à função get_monitorable da API C, invocada com o argumento *id* preenchido. Ele retorna uma coleção de todos os objetos do tipo MonitorableProcess associados ao objeto *MonitorableMachine* informado no argumento *machine*.

- **Monitorable findMonitorable(String id)** Este método equivale à função *find_monitorable* da API C. Ele retorna o objeto Monitorable cujo identificador seja igual ao argumento *id*.

Notifier

Permite que processos ou objetos sejam configurados para serem notificados da ocorrência de eventos pelo serviço. Seus métodos repassam as invocações para as funções do grupo Notificação da API C. A seguir, a descrição de seus métodos.

- **void addMonitoringNotifiable(MonitoringNotifiable notifiable)** Este método equivale à função *add_monitoring_notifiable* da API C. Seu único argumento, *notifiable*, contém todos os dados necessários para adicionar um notificável no serviço. O notificável pode ser tanto uma instância de *MonitoringNotifiableProcess*, que representa um processo, quanto qualquer objeto que implemente a interface *MonitoringNotifiableObject*. Neste caso, a notificação ocorrerá seguindo o padrão de projeto *Observer*, também adotado na API Java, através da invocação de métodos específicos para cada evento.

- **void addMonitoringNotifiable(String id, Map events)** Esta versão sobrecarregada do método anterior deve ser utilizada quando o processo atual quiser se cadastrar

como notificável de eventos. O *pid* do processo corrente será utilizado para identificar o processo local a ser cadastrado como notificável e o argumento *id* informa o identificador no domínio de detecção pretendido para ele. O argumento *events* serve para relacionar cada evento a um sinal do sistema operacional que deverá ser enviado para o processo em caso de notificação. Os eventos são identificados através de objetos definidos como atributos estáticos da classe MonitoringEvent do pacote *br.edu.ufcg.dsc.delphus.event*: *MONITORABLE_FAILED*, *MONITORABLE_LEFT* e *MONITORABLE_JOINED*. Os sinais são identificados através de objetos definidos como atributos estáticos da classe *LinuxSignals*, do pacote *br.edu.ufcg.dsc.delphus.util*: *SIGINT*, *SIGCHLD*, etc.

- **void removeNotifiable(String id)** Este método equivale à função *remove_notifiable* da API C.

- **void removeNotifiable(Notifiable notifiable)** Esta versão sobrecarregada do método anterior permite que o objeto a ser removido seja identificado por um objeto *Notifiable*, ao invés de diretamente por seu identificador.

- **boolean isCorrect(String id, java.util.Date time)** Este método equivale à função *is_correct* da API C.

- **boolean isFaulty(String id, java.util.Date time)** Este método equivale à função *is_faulty* da API C.

- **Collection getCorrect()** Este método é equivalente à função *get_correct* da API C. Retorna uma coleção de objetos *Monitorable* que não tenham falhado.

- **Collection getFaulty()** Este método é equivalente à função get_faulty da API C. Retorna uma coleção de objetos *Monitorable* que já tenham falhado.

- **Collection getMonitoringNotifiable()** Este método equivale à função *get_monitoring_notifiable* da API C, invocada com o argumento *monit_id* igual a nulo. Ele retorna uma coleção de todos os objetos notificáveis de monitoramento existentes.

- **Collection getMonitoringNotifiable(String monitorableId)** Este método é equivalente à função *get_monitoring_notifiable* da API C, invocada com o argumento *monit_id* informado. Ele retorna uma coleção de todos os objetos notificáveis interessados na ocorrência dos eventos com o monitorável identificado pelo argumento *monitorableId*.

- **MonitoringNotifiable findMonitoringNotifiable(String id)** Este método é equivalente à função *find_monitoring_notifiable* da API C. Ele retorna o objeto notificável, cujo identificador seja igual ao argumento *id*.

Para permitir um tratamento de erros mais elaborado do que o possível com a API C, foram criadas exceções que podem ser produzidas pelos métodos vistos acima. Estas exceções estão relacionadas aos códigos de erro vistos na API C, conforme pode ser visto na Tabela 9. O diagrama de classes com a hierarquia das exceções pode ser vista na Figura 39. A descrição de cada classe de exceção encontra-se na Tabela 9. As situações em que cada uma ocorre no serviço são as mesmas descritas na Seção 4.3.1.

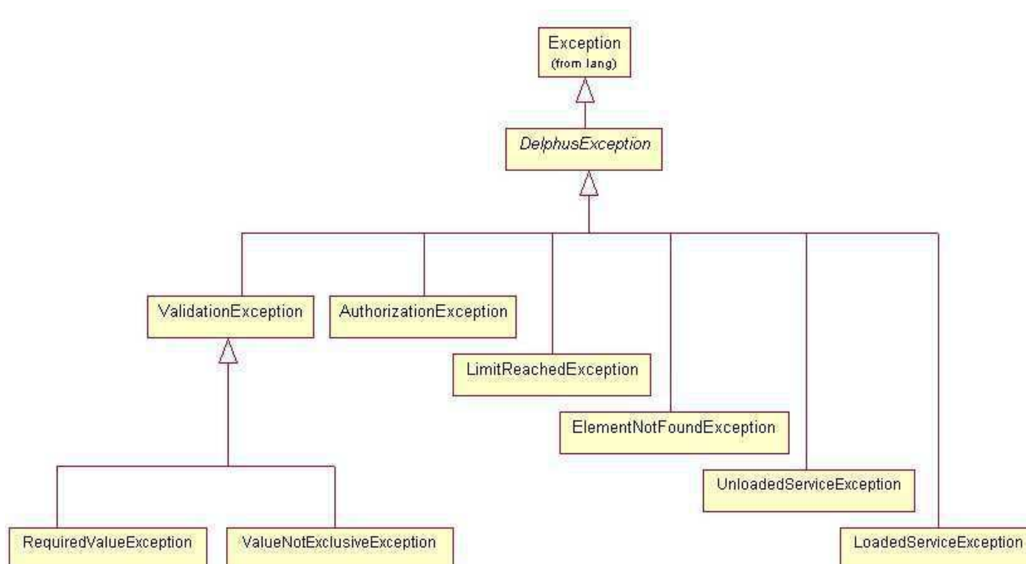


Figura 39: Exceções da API Java

Exceção	Descrição	Erro relacionado
UnloadedService	*O serviço não foi carregado ainda*	UNLOADED_SERVICE_ERROR
LoadedService	*O serviço já foi carregado*	LOADED_SERVICE_ERROR
AuthorizationException	*Usuário não autorizado a executar método*	AUTHORIZATION_ERROR
LimitReachedException	*Não é possível inserir mais um ítem*	LIMIT_REACHED_ERROR
ElementNotFoundException	*Elemento não pôde ser encontrado*	ELEMENT_NOT_FOUND_ERROR
ValidationException	*Erro de validação de valor*	VALIDATION_ERROR
RequiredValueException	*Valor requerido não informado*	REQUIRED_VALUE_ERROR
ValueNotExclusiveException	*Valor não exclusivo*	VALUE_NOT_EXCLUSIVE_ERROR

Tabela 9: Exceções da API Java

4.3.3 Ferramenta de Administração Olen

A ferramenta Olen[9] foi implementada com o objetivo de fornecer um auxílio para a administração de um domínio de detecção. Construída sobre a API em Java, ela oferece uma interface gráfica amigável para uso do serviço. Todos os objetos do domínio de detecção são organizados hierarquicamente em uma estrutura de árvore na parte esquerda da janela da aplicação. Ao selecionar um destes elementos, suas propriedades são exibidas em uma tabela à direita. Mensagens sobre a execução das operações do usuário são exibidas na parte inferior direita da janela, com o horário em que foram executadas.

A Figura 40 apresenta uma janela do Olen em um domínio com duas máquinas, dois processos sendo monitorados e quatro processos notificáveis. Os ícones *m1* e *m3* representam as máquinas do domínio de detecção. Os ícones *p1* e *p2* representam dois processos monitoráveis, em execução em *m3*. Todos os ícones de máquinas e processos monitoráveis possuem uma pasta chamada *notificáveis*, que agrupa os processos notificáveis interessados em notificação de seus eventos. Os ícones *n1* e *n2*, por exemplo, representam processos notificáveis, em execução em *m3* que devem ser notificados da ocorrência de eventos com o monitorável *p2*. Existe uma pasta no final da árvore, não associada a monitorável algum, que agrupa os processos notificáveis de eventos com qualquer entidade monitorável. Como exemplo, pode se ver o notificável *_1* que representa o objeto *Notifier* da interface Java (explicado em detalhes na Seção 4.3.2). Ele é responsável por fazer o Olen receber as notificações, que o

[9] Olen foi um dos sacerdotes de Apolo no oráculo de Delphus. Ele era um dos intermediários entre o povo e o oráculo.

permitem oferecer sempre uma visão consistente da configuração do domínio.

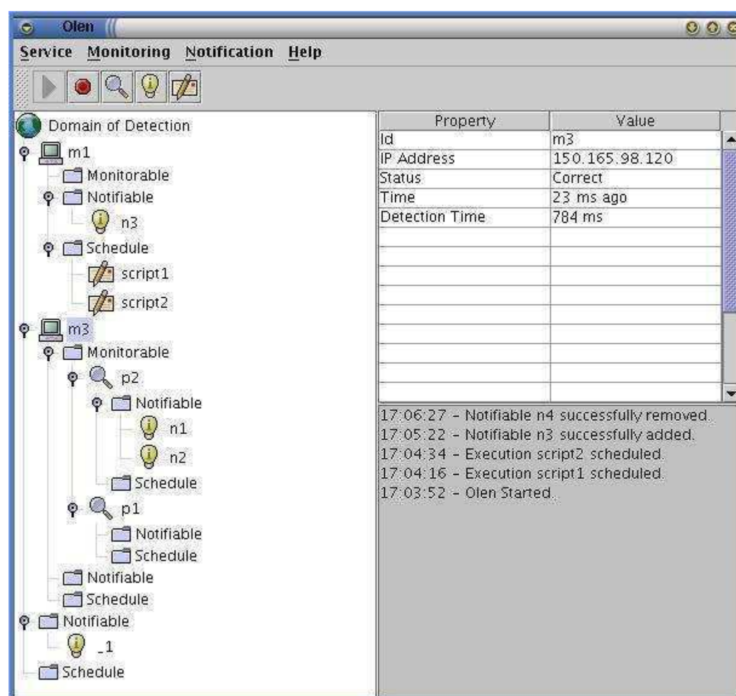


Figura 40: Olen - Ferramenta de Administração de Domínio de Detecção

Através de seus menus e barra de ferramentas, é possível cadastrar um processo já existente como monitorável no serviço, ou executar um programa e ter o novo processo criado cadastrado para monitoria automaticamente. Também é possível cadastrar um processo já existente como notificável no serviço, ou executar um programa e ter o novo processo criado cadastrado para notificação, automaticamente.

Além disso, é possível agendar execuções de programas ou *scripts* para quando determinados eventos ocorrerem. A Figura 40 mostra dois *scripts: script1* e *script2* programados para serem executados quando uma falha for detectada na máquina *m1*.

Um exemplo da utilização desta ferramenta pode ser visto na Seção 5.

4.3.4 Comandos de Console

Vários comandos de console foram criados para disponibilizar uma interface a caracter para uso e administração do serviço e permitir a criação de *scripts*. A seguir, estes comandos

são descritos em detalhes. Alguns deles possuem restrições a respeito dos usuários que têm permissão para utilizá-los. Quando essas restrições não forem explicitadas, deve-se assumir que elas não se aplicam ao comando. Qualquer falha na execução destes comandos será informada ao usuário através de uma mensagem na tela. Estes comandos foram desenvolvidos usando a API C do serviço e a maioria possui relação direta com suas funções.

- **startm** *asyncip syncip periodtime* – Este comando deve ser usado por um administrador para iniciar o serviço em uma máquina. Seu efeito é carregar o módulo Linux do Delphus. Os argumentos *syncip* e *asyncip* informam os endereços IP a serem utilizados pelo o módulo do serviço para acessar as redes síncrona e assíncrona, respectivamente. O argumento *periodtime* informa a duração de um período TDMA (em milisegundos), pretendido para o domínio de detecção. A invocação deste comando equivale a usar o comando *insmod* para carregar o módulo do Delphus.

- **stopm** – Este comando deve ser usado por um administrador para interromper o serviço em uma máquina e consequentemente, remover todas as suas entidades monitoráveis e ela própria do domínio de detecção. Como resultado da execução deste comando, o módulo do Delphus é descarregado do sistema.

- **isdstarted** – Este comando pode ser utilizado para saber se o serviço foi iniciado na máquina. Uma mensagem informará se o módulo do Delphus está ativo ou não.

- **addmonit** *id pid detectiontime* – Este comando serve para configurar um processo local para ser monitorado. Ele só poderá ser usado por usuários proprietários dos processos, ou pelo super usuário. O argumento *id* informa o identificador no domínio para o processo e o *pid* identifica o processo no sistema operacional. O *detectiontime* informa em quanto tempo (expresso em milisegundos), falhas no processo deverão ser detectadas pelos módulos do serviço.

- **rmmonit** *id* – Quando uma entidade tem que ser removida do domínio de detecção, este comando deve ser usado. O argumento *id* identifica a entidade a ser removida. Este comando também só pode ser usado pelo usuário proprietário do processo a ser removido, ou pelo super usuário.

- **lsmonit** – Para ver uma lista completa das entidades monitoráveis do domínio de detecção, este comando deve ser usado. Os processos são agrupados por máquina e todos os atributos dos monitoráveis são exibidos, como *pid*, endereço IP e estado (correto ou falho).

- **findmonit** *id* – Este comando deve ser usado quando se quer ver as propriedades de uma entidade monitorável específica, ou quando se quer saber se um monitorável existe. Ele localiza a entidade por seu *id* e lista suas propriedades.

- **lscorrect** – Este comando lista as entidades monitoráveis detectadas como corretas até o momento.

- **lsfaulty** – Este comando lista as entidades monitoráveis detectadas como falhas até o momento.

- **addnotif** *type id pid monitorableid failuresignal leavesignal joinsignal* – Este comando adiciona um processo notificável no domínio de detecção. O argumento *type* identifica o tipo de notificação pretendido. Como atualmente o único disponível é o de monitoramento, o único valor disponível para este argumento é *m*. O argumento *id* informa a identificação pretendida para o processo notificável no serviço e o argumento *pid* identifica o processo no sistema operacional. O argumento *monitorableid* informa o identificador da entidade monitorável no domínio de detecção, da qual se quer a notificação de eventos. Este comando só poderá ser usado por usuários proprietários dos processos, ou pelo super usuário. Os argumentos *failuresignal*, *leavesignal* e *joinsignal* informam os sinais do Linux que deverão ser enviados para o processo notificável, caso ocorram os eventos falha em monitorável, remoção de monitorável e admissão de monitorável, respectivamente.

- **rmnotif** *id* – Este comando deve ser utilizado para remover uma entidade notificável. O argumento *id* identifica a entidade a ser removida. Este comando também só pode ser usado pelo usuário proprietário do processo a ser removido, ou pelo super usuário.

- **lsnotif** – Este comando deve ser usado para ver uma lista completa das entidades notificáveis do domínio de detecção. Os processos são agrupados por categoria de evento. Como atualmente existe apenas a categoria de monitoramento, todos os notificáveis são reunidos em um único grupo. Os atributos dos notificáveis como *pid* e *monitorableid* são listados, assim como, no caso dos notificáveis de monitoramento, os sinais associados aos eventos de monitoramento: *failuresignal, leavesignal, joinsignal.*

- **findnotif** *id* – Este comando deve ser usado quando se quer ver as propriedades de uma entidade notificável específica, ou quando se quer saber se ela existe. Ele localiza a entidade por seu *id* e lista suas propriedades.

5 Usando o Serviço

Nesta seção, apresentaremos um caso popular de utilização do Delphus. O exemplo consiste em um serviço web, com alto grau de disponibilidade, oferecido em uma rede como a Internet.

Para garantir a disponibilidade do serviço, técnicas de tolerância a falhas devem ser empregadas no sistema, para que falhas não interrompam o seu fornecimento.

Uma técnica bastante comum que poderia ser adotada é a da replicação passiva seguindo a abordagem primário-cópia. Uma máquina seria o servidor primário, inicialmente configurada para fornecer o serviço na rede. Outra máquina seria o servidor cópia, capaz de fornecer o mesmo serviço e pronta para assumir o endereço IP da primeira, em caso de falha.

A adoção desta técnica tornaria transparente aos usuários do serviço eventuais falhas no servidor primário. Entretanto, para que possa ser implementada de forma consistente, esta técnica necessita de um serviço de detecção de falhas. Este serviço permitiria que o servidor cópia se tornasse ciente da falha no servidor primário quando esta ocorresse e assim pudesse assumir o seu lugar.

O serviço de detecção de falhas não poderia cometer erros, pois isto levaria o servidor cópia a se considerar primário, sem que o primeiro tivesse realmente falhado. Isso geraria conflitos que interromperiam o fornecimento do serviço web. Além do mais, esta detecção deveria ser a mais rápida possível, de modo que a reconfiguração no ambiente não fosse percebida pelos usuários.

Esses requisitos e o fato de que estes servidores normalmente executam em ambientes controlados, em redes locais com pouca dispersão geográfica, tornam o Delphus uma ferramenta de suporte apropriada para implementação da replicação.

Para ilustrar a execução do exemplo, consideramos três máquinas. Duas são utilizadas para o serviço (*m1*, *m2*) e uma como cliente (*m3*). A máquina *m1* é o servidor primário e a máquina *m2* é o servidor cópia. Nas máquinas servidoras, a interface *eth0* implementa o canal de comunicação de uso exclusivo do serviço de detecção de falhas. A interface de rede *eth1* é utilizada para a comunicação padrão com o restante da rede.

A interface *eth1* foi escolhida para a comunicação padrão com a rede, porque quando ativada, o Linux cria uma *kernel thread* para gerenciá-la com a descrição “eth1”, facilmente

identificável na lista de processos retornada pelo comando *ps*. Isto facilita a consulta de seu *pid*, necessário nos passos seguintes deste exemplo. A interface de rede *eth0* por sua vez, não cria uma *kernel thread* na tabela de processos quando é ativada. Isso dificulta seu monitoramento pelo Delphus.

No caso de ocorrer uma falha por parada na *kernel thread eth1* na máquina *m1*, ou na própria máquina *m1*, ou ainda esta interface ser desativada com o comando *ifconfig eth1 down*, a interface de rede *eth1* de *m2* é reconfigurada para responder pelo endereço IP de *eth1* em *m1*. A máquina *m3* serve como cliente web, e frequentemente acessa o serviço.

A ferramenta de administração Olen, parte da interface Java do Delphus, pode ser utilizada para iniciar o Delphus nas máquinas, adicionar o servidor *m1* para ser monitorado, e programar um *script* para ser executado, quando uma notificação de falha no servidor fosse recebida. Este *script* será capaz de configurar o servidor cópia *m2* para assumir o endereço IP do servidor primário *m1*. Secundariamente, este *script* poderá alertar o administrador do sistema, enviando um e-mail para o mesmo, por exemplo. Na Figura 41, pode ser visto um exemplo deste *script*.

```
ifconfig eth1 150.165.98.39 broadcast 150.165.98.127 netmask 255.255.255.0
mail -s Falha no servidor primário administrador@dominio.com < /scripts/alerta.txt
```

Figura 41: Script *activate_backup* de configuração do servidor primário

A primeira linha possui o comando que configura *m2* para assumir o endereço IP de *m1*: 150.165.98.39. A segunda linha usa o comando *mail* para enviar um e-mail com o assunto "Falha no servidor primário" para o usuário administrador, cujo endereço de e-mail é *administrador@dominio.com*. O corpo do e-mail é especificado pelo arquivo *alerta.txt*, localizado no diretório */scripts*.

O primeiro passo para implantar esta técnica, é iniciar o serviço nas máquinas *m1* e *m2*. A Figura 42 mostra este passo sendo tomado na máquina *m1*.

A Figura 43 mostra o instante em que as duas máquinas possuem o serviço iniciado.

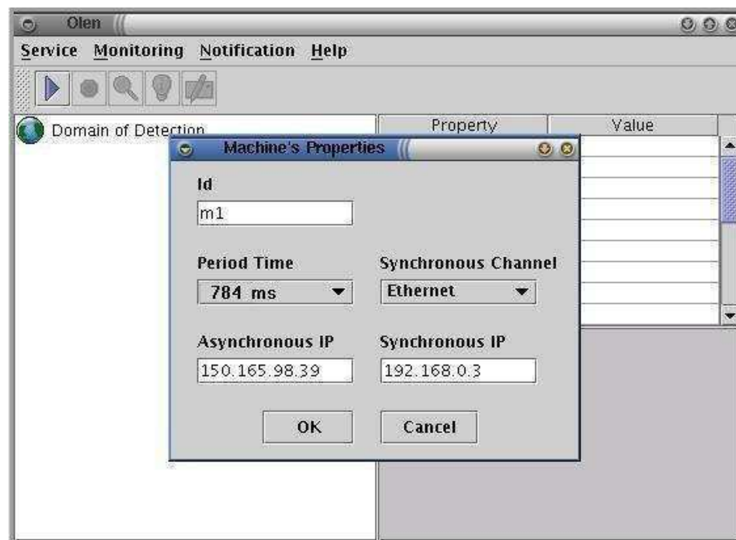


Figura 42: Iniciação do serviço na máquina m1

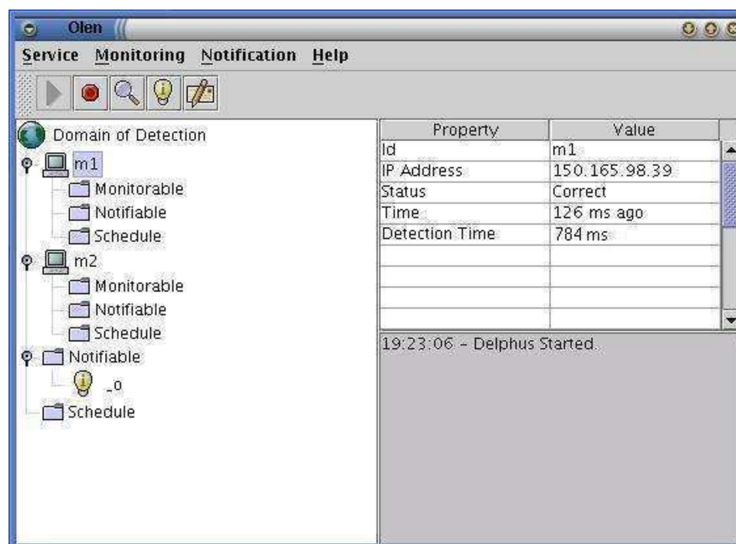


Figura 43: Serviço iniciado nas 2 máquinas

Depois, deve-se adicionar a *kernel thread eth1* de *m1* como monitorável. A Figura 44 mostra o instante em que *a kernel thread eth1* da máquina *m1*, cujo pid é igual a 550, é cadastrada como monitorável, com o identificador *p1* e tempo máximo para detecção de falhas igual a 784 ms.

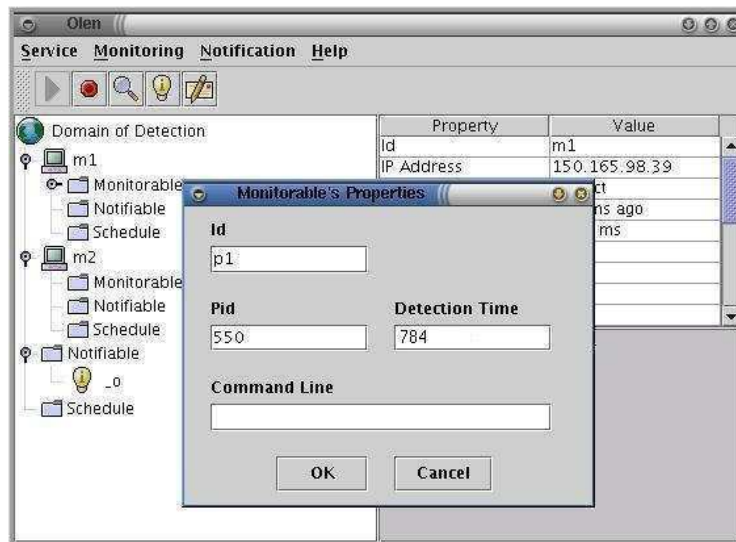


Figura 44: Cadastro da kernel thread eth1 como monitorável

A Figura 45 mostra o instante em que o *script activate_backup*, visto na Figura 41, é cadastrado para ser executado, no momento em que o Olen receber uma notificação de falha em *p1*. A Figura 46 mostra o *script* já cadastrado em *m2*.

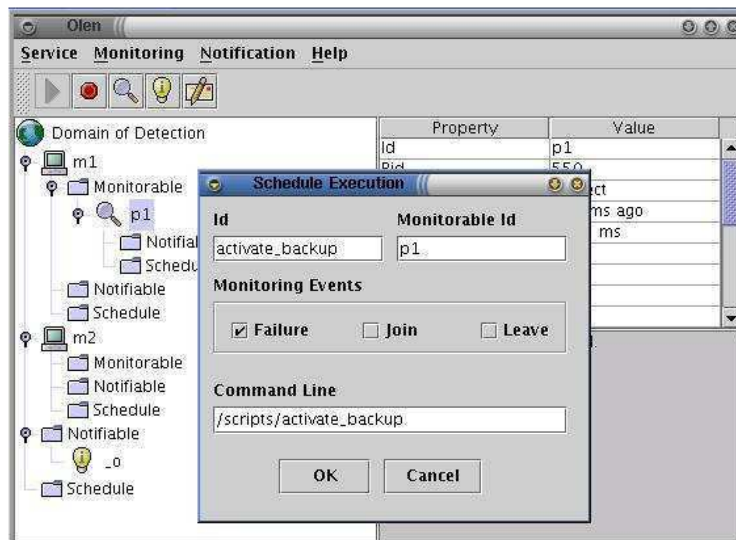


Figura 45: Cadastro do script activate_backup

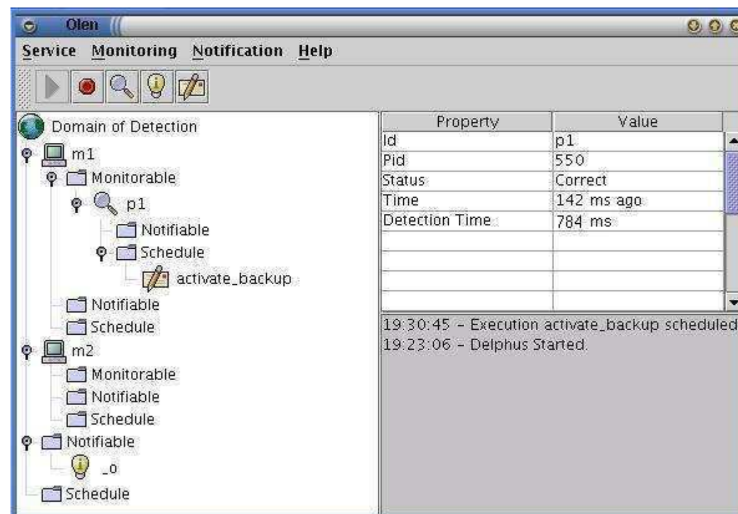


Figura 46: Script activate_backup cadastrado

Estando o serviço configurado, ou seja, o servidor primário em operação e o servidor cópia monitorando o funcionamento do primário, o serviço pode ser testado. Da máquina *m3*, será possível acessar o serviço web, mesmo que *m1* seja abruptamente desligada, ou que sua *kernel thread eth1* seja terminada através da execução do comando *ifconfig eth1 down*, por exemplo. Em no máximo 784 *ms*, o Delphus detectará a falha na *kernel thread eth1* de *m1* e enviará um sinal de notificação para o Olen, em execução no servidor cópia. Este, por sua vez, executará o *script activate_backup* que fará a interface *eth1* assumir o endereço de *m1*. Isto fará a máquina *m2* assumir o serviço antes prestado por *m1*, tornando a falha em *m1* transparente para os clientes do serviço.

Este foi um exemplo simples de utilização do Delphus. Entretanto, é possível a partir dele, estruturar um ambiente mais complexo, com mais servidores, e outros serviços sendo oferecidos.

6 Avaliação de Desempenho do Serviço

Durante a fase de projeto do Delphus, seus protocolos foram modelados formalmente em redes de petri coloridas. Através da ferramenta Design CPN [1], simulações foram realizadas para testar a sua corretude, antes de serem implementados. Isto deu maior garantia de que o serviço se comportaria conforme sua especificação.

Durante a fase de implementação do serviço, foram feitos testes de unidades com a ferramenta JUnit [5]. A estratégia utilizada foi de testar as funcionalidades em alto nível do serviço, através da API Java. Como a API C é utilizada pela API Java, os testes acabam envolvendo todos os níveis da implementação. Infelizmente, ainda não existem ferramentas de teste de unidade voltadas para ambientes distribuídos. Sendo assim, funcionalidades como eleição de líder, tiveram que ser testadas da maneira tradicional, sem a automatização de testes. No entanto, a maior parte das funcionalidades do serviço tiveram seus testes automatizados, o que facilita novas alterações e ajuda a aumentar a qualidade do software.

Uma preocupação natural para um usuário do Delphus é o quanto o sistema seria afetado com a presença deste serviço. O fato de que suas *threads* possuem prioridade de tempo real, sempre acima das de qualquer outro processo em execução nas máquinas, provoca uma certa apreensão quanto aos seus possíveis efeitos negativos sobre desempenho do sistema.

Em primeiro lugar, é necessário considerar que versões genéricas do sistema operacional não oferecem suporte para aplicações com requisitos temporais. Desta forma, as aplicações projetadas para este ambiente não podem fazer nenhuma suposição sobre o tempo que gastarão para executar suas tarefas, nem sobre quando elas serão executadas. O uso da CPU é compartilhado por todas as aplicações em execução na máquina e é controlado pelo escalonador do sistema operacional. Sendo assim, o fato de que as *threads* do Delphus poderão atrasar a execução destas aplicações, não é por si só, uma característica negativa do serviço. Outras aplicações poderiam fazê-lo, mesmo sem ter prioridade de tempo real.

Mesmo assim, o Delphus foi desenvolvido para reduzir ao máximo a interferência na execução das aplicações e consumir o mínimo possível de recursos do sistema. Para quantificar este impacto, alguns testes foram realizados. Foram usadas 5 máquinas Pentium IV, com 632 *Mbytes* de memória RAM, clock de 1.7 *GHz*, disco rígido de 40 *GBytes*, conectados

a uma rede Ethernet de 100 *Mbps*. A distribuição do Linux usada foi a Mandrake Linux 9.0, kernel versão 2.4.19, configurados para uma taxa de 1000 interrupções de relógio por segundo. Nestas máquinas não foram carregados serviços como servidores web, SSH, Servidor X, para que não interferissem nos resultados dos testes.

6.1 Total de memória utilizada pelo serviço

O primeiro teste realizado teve o objetivo de medir o quanto de memória RAM é consumida pelo módulo do serviço, quando este atinge sua carga máxima, assim como a taxa de evolução deste consumo em função do aumento da carga.

O primeiro passo foi medir a memória consumida nas máquinas, após serem iniciadas. Usando o comando *top*, os valores foram coletados. Um cuidado foi tomado: estes valores foram observados por alguns minutos, para que se tivesse certeza de que permaneciam estáveis, indicando que nenhuma atividade na máquina aumentava o consumo de memória. Depois, o serviço foi iniciado em uma máquina e o novo valor de memória consumida foi coletado no *top*. Em seguida, o serviço foi interrompido naquela máquina e o mesmo procedimento foi realizado nas outras máquinas. Ao final da coleta destes dados, foi possível calcular uma média da memória gasta nas máquinas com a simples iniciação do serviço: 624 *Kbytes*.

O segundo passo foi, em uma máquina com o serviço iniciado, adicionar o máximo de processos monitoráveis (50) e medir a memória consumida na máquina. As medições foram realizadas a cada inserção de 10 entidades. Os resultados destas medições podem ser vistos no gráfico da Figura 47.

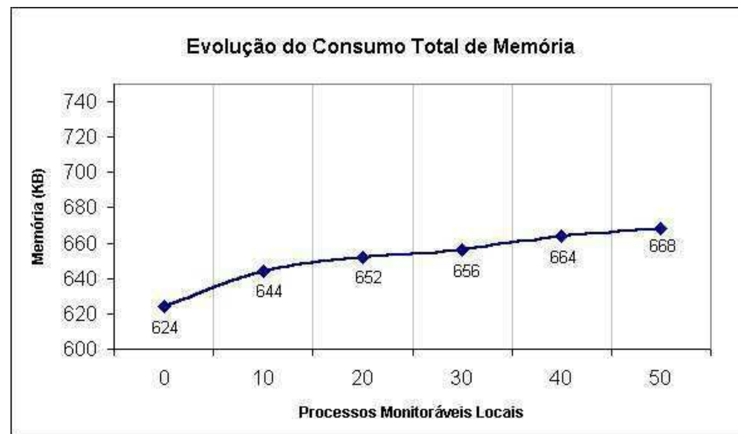


Figura 47: Consumo Total de Memória - Monitoráveis Locais

O primeiro grupo de 10 monitoráveis adicionados, fez crescer em 20 *Kbytes* a memória em uso na máquina. Entretanto, logo após isso, os demais monitoráveis aumentaram em média, 6 *Kbytes* este espaço. Isso dá uma média de uso da memória de 600 *bytes* para cada monitorável, desconsiderando o espaço excepcional ocupado pelo primeiro grupo. Ao final, a memória total ocupada pelo serviço aumentou para 668 *Kbytes*.

O terceiro passo foi iniciar o serviço nas outras quatro máquinas, adicionar o máximo de processos monitoráveis (50) e medir a evolução do espaço ocupado na primeira máquina. O resultado pode ser visto no gráfico da Figura 48.

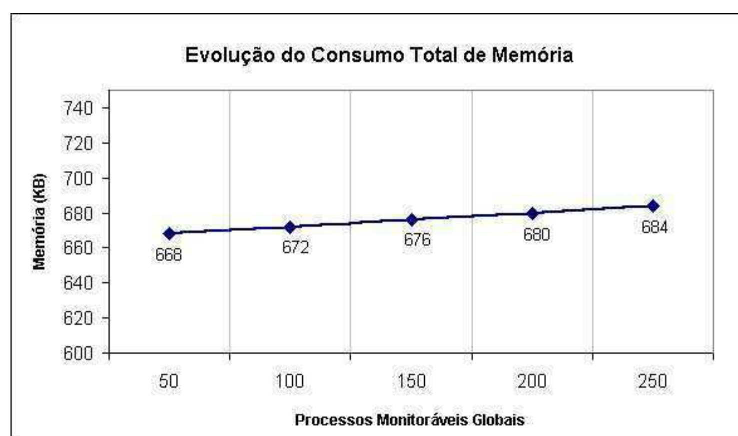


Figura 48: Consumo Total de Memória - Monitoráveis Globais

Cada máquina que ingressou no domínio com seus monitoráveis, provocou um aumento de 4 *Kbytes* no total de memória em uso na primeira máquina, o que representa uma média de menos de 80 *bytes* por monitorável.

O último passo foi adicionar na primeira máquina o máximo de processos notificáveis permitido, que é de 100, em grupos de 20 e medir a evolução do consumo de memória. Os resultados podem ser vistos no gráfico da Figura 49.

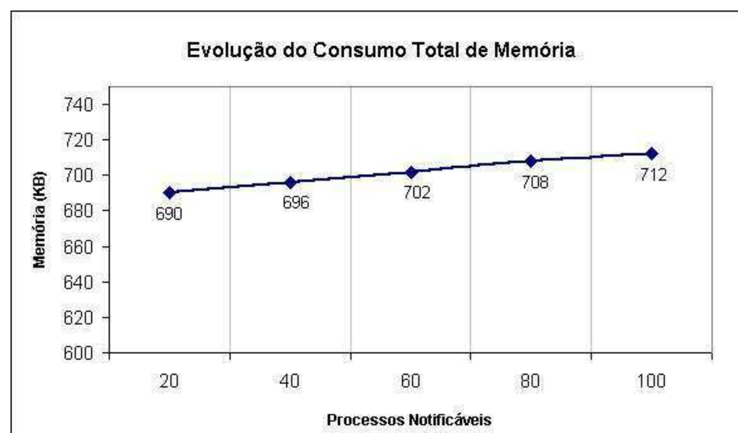


Figura 49: Consumo Total de Memória - Notificáveis

Cada notificável adicionado provocou um aumento de 300 *bytes* no total de memória consumida. Ao final dos testes, com a carga máxima de utilização do serviço, o total de memória consumida na primeira máquina, líder do domínio e consequentemente a mais carregada de todas foi de 712 *Kbytes*. Este valor representa aproximadamente 0.11% do total de memória RAM das máquinas usadas no teste, que equivale a uma parcela muito pequena e de pouco impacto para o sistema.

Os gráficos também revelam uma progressão aritmética do consumo de memória. Isto facilita a previsão do total de memória consumida em um domínio de detecção com mais máquinas e demonstra que o serviço possui escalabilidade.

Ao final, a memória em uso foi medida nas demais máquinas e foi constatado que, o espaço ocupado pelo serviço teve uma média de 711 *Kbytes*. Depois, o serviço foi deixado em execução nas máquinas por 20 minutos e não foi observado um aumento da memória em uso. Isso revela a ausência de vazamento de memória com a instalação do Delphus.

6.2 Percentual de ocupação da CPU pelo serviço

O segundo teste realizado teve o objetivo de medir o quanto do tempo de CPU é utilizado pelo serviço para executar suas *threads* de tempo real, no pior caso. Com a mesma carga utilizada no teste anterior, o tempo total gasto pela CPU na execução das *threads* foi medido em cada máquina. Esta medição foi feita através da adição no código do serviço de chamadas à função *do_gettimeofday* (que retorna o tempo com precisão de microsegundos) no início e final de cada trecho de código em que as *threads* estariam em execução na CPU, e não dormindo. Usando esta função, foi possível calcular a duração de cada execução e acumular o tempo em uma variável. Outra variável acumulava o número de períodos TDMA passados. Estas variáveis eram consultadas a cada 30 minutos e seus valores eram usados para calcular o percentual de utilização da CPU pelo Delphus até aquele instante. Este procedimento foi realizado em duas situações: o sistema configurado para 100 e configurado para 1000 interrupções de relógio por segundo. Na primeira situação, o tamanho do período TDMA foi de 966 ms e na segunda, foi de 784 ms. Estes tamanhos são os menores possíveis e os que representam as maiores cargas para o sistema, em cada configuração.

Os resultados obtidos podem ser vistos nos gráfico da Figura 50.

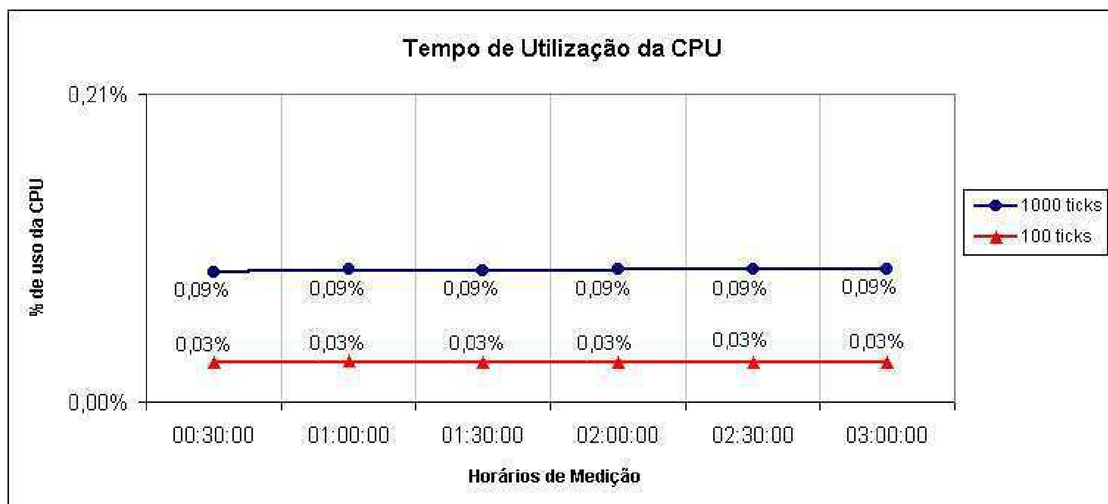


Figura 50: Percentual de Ocupação da CPU em 3 horas

No sistema configurado para 1000 interrupções de relógio por segundo, a média de utili-

zação da CPU foi de $0,09\%$. Já no de 100 interrupções de relógio por segundo, a média foi de $0,03\%$. Estes valores demonstram que durante uma parcela ínfima de tempo, o Delphus terá acesso à CPU. Este valor é certamente muito inferior ao usado por outros programas e serviços do sistema, que não possuem prioridade de tempo real. Isso ocorre porque na maior parte do tempo, as *threads* do serviço estão dormindo, ou bloqueadas, esperando a chegada de mensagens. O tempo em que elas estão prontas para executar e recebem tratamento privilegiado pelo escalonador de processos é muito pequeno.

6.3 Sobrecarga da rede

Foi avaliado também o quanto o tráfego de mensagens pela rede era aumentado, com o uso do Delphus. Convém lembrar que a maior parte das mensagens do serviço trafega pela rede síncrona, de uso exclusivo pelo sistema, não afetando portanto, a carga da rede usada pelo restante do sistema. As poucas mensagens que trafegam pela rede assíncrona são trocadas durante o processo de admissão de monitoráveis e iniciação do serviço em uma máquina. A maior delas é a mensagem que contém a lista de todos os monitoráveis do domínio de detecção, que é enviada pelo líder à máquina, durante seu ingresso no domínio. Com a carga máxima do Delphus, usada nos testes anteriores, esta mensagem tem o tamanho máximo inferior a 60 *Kbytes*, considerando apenas os dados, sem levar em conta o espaço gasto pelos protocolos de rede para transmiti-la. Além de serem pequenas, o fato de que só são transmitidas esporadicamente, torna irrelevante o efeito que teriam sobre a carga da rede.

6.4 Teste de desempenho geral do sistema

Apenas com os resultados dos testes anteriores, é possível concluir que o Delphus não representa um fator de degradação do desempenho do sistema.

Mesmo assim, um teste geral do desempenho do sistema foi realizado, utilizando a ferramenta de *benchmark* UnixBench. O Unixbench 4.01 mede o desempenho de operações de IO e execuções multitarefa no sistema operacional. Ele é adotado pela revista Byte Magazine para realizar testes que são usados em suas matérias. O Unixbench é composto por uma série de testes, compartilhados com outras ferramentas de *benchmark*. Maiores informações

sobre o Unixbench podem ser obtidas no site www.tux.org. A Tabela 10 descreve cada um destes testes.

Teste	Descrição
Cópia de arquivos	Taxa de transferência de dados entre dois arquivos, usando vários tamanhos de buffer
Pipe Throughput	Taxa de transmissão de dados entre dois processos, via pipe
Execl Throughput	Número de system calls execl executadas por segundo
Shell Scripts	Eficiência na execução de scripts de comandos de console
Criação de processos	Quantidade de processos criados por segundo
System Call Overhead	Número de system calls executadas por segundo
Dhrystone 2	Eficiência de operações aritméticas
Whetstone	Eficiência de operações aritméticas
Byte Final Score	Pontuação final atribuída ao desempenho do sistema testado

Tabela 10: Testes que compõem o Unixbench

Inicialmente, a bateria de testes do Unixbench foi executada duas vezes em todas as 5 máquinas usadas nos testes, antes do Delphus ser iniciado. Era esperado que estas máquinas apresentassem resultados próximos de desempenho, uma vez que possuiam a mesma especificação e configuração. Uma análise estatística realizada sobre estes dados revelou que os testes *Execl Throughput* apresentaram o maior desvio padrão dentre todos os testes. Este desvio foi dez vezes superior à média dos demais, indicando que algumas máquinas obtiveram comportamento muito diferenciado das demais. Isto indicou a interferência de fatores desconhecidos nestes resultados. Este teste não foi então utilizado, para que estes fatores desconhecidos não levassem a conclusões erradas. Apenas os testes onde as máquinas obtiveram desempenhos próximos foram considerados neste experimento.

Depois, com o Delphus executando com carga máxima, o Unixbench foi novamente executado duas vezes. O sistema operacional estava configurado para 1000 interrupções de relógio por segundo, que é a configuração em que o Delphus ocupa mais tempo de CPU, conforme observado nos testes anteriores. Conforme explicado anteriormente, nenhum programa ou serviço que pudesse interferir nos testes estava sendo executado, as máquinas possuiam a mesma configuração e estiveram completamente dedicada aos testes. Novamente os resultados foram analisados estatisticamente, e dois testes obtiveram desvio padrão muito alto (10 vezes superior aos demais): *Execl Throughput* e *Shell Scripts*. Pelos mesmos motivos

explicados anteriormente, os resultados destes testes não foram considerados.

Os resultados finais de variação de desempenho podem ser vistos no gráfico da Figura 51.

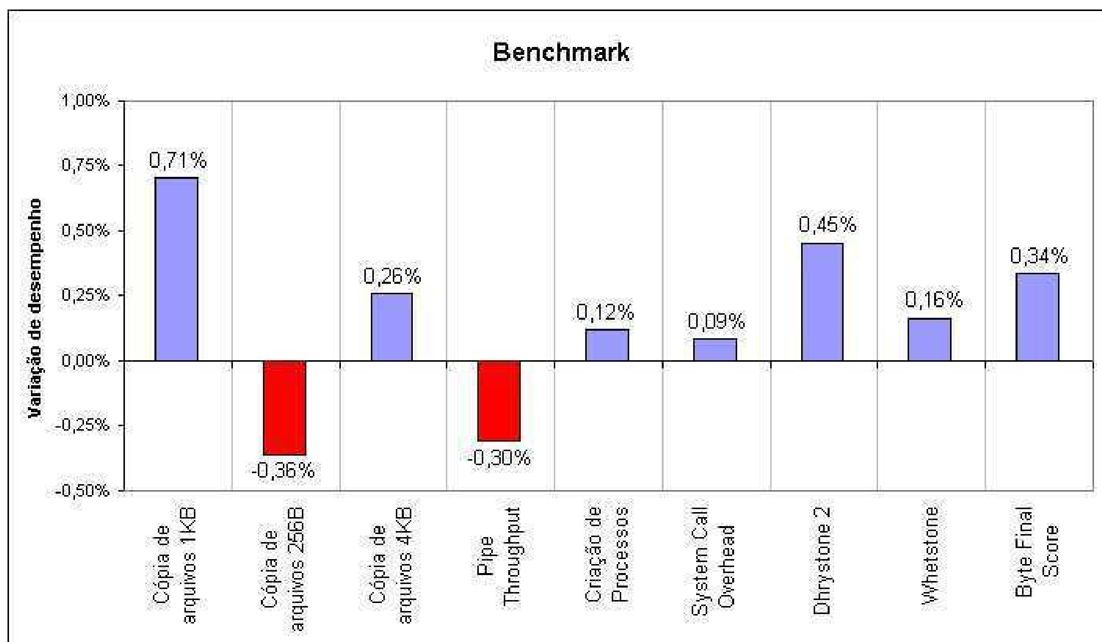


Figura 51: Benchmark do Sistema com o Delphus

O gráfico informa o percentual de variação de desempenho identificado por cada teste do benchmark. Ele revela que houve uma variação muito baixa do desempenho do sistema antes e depois da instalação do Delphus. Alguns testes revelaram até uma variação positiva do desempenho das máquinas após a instalação do serviço. Obviamente, isso não pode ser traduzido como um benefício trazido pela adoção do serviço. Isso se deve a outras variáveis do ambiente, não identificadas, que afetaram positivamente o desempenho das máquinas. Entretanto, mesmo com a existência destas interferências, observa-se que os percentuais de variação entre os dois momentos são inferiores a $0,75\%$. Mesmo que tivessem sido negativos e pudessem ser atribuídos à presença do Delphus, ainda assim não revelaria uma perda significativa de desempenho.

7 Conclusões

7.1 Análise do Trabalho

Ao final deste trabalho, é possível concluir que o seu objetivo - projetar e implementar um serviço de detecção a falhas com semântica perfeita, com garantias de qualidade de serviço, exigindo poucas restrições do sistema - foi alcançado. O Delphus traz uma importante contribuição para a área de tolerância a falhas em sistemas distribuídos, uma vez que vem suprir uma demanda da área por serviços com tais características. Várias aplicações requerem serviços de detecção de falhas que não cometam erros e a falta de opções levou tais aplicações a adotarem mecanismos que apresentam janelas de vulnerabilidade.

Alguns serviços com semântica perfeita já foram implementados. Entretanto, eles requerem a adoção de fortes restrições ao ambiente, que nem todos os sistemas têm condições de adotar. Além disso, alguns requerem um alto grau de acoplamento com as aplicações clientes e outros possuem restrições de escalabilidade. O Delphus propõe-se a servir sistemas de propósito geral, que não precisam ou não podem adotar fortes restrições em seus ambientes, como é o caso da maioria dos sistemas práticos atuais.

A análise de desempenho do Delphus revelou ainda que sua atividade traz uma influência desprezível no desempenho do sistema. Isto reduz ainda mais as restrições impostas aos sistemas clientes.

7.2 Considerações Finais

Chandra e Toueg provaram que a classe $\diamond S$ é a mais fraca classe de detectores de falhas que permite uma solução para o problema do consenso [17]. Influenciadas por este resultado, inúmeras implementações de detectores de falhas $\diamond S$ foram tratadas na literatura e vários protocolos de alto nível foram construídos com o uso de tais implementações. Devido à semântica da classe $\diamond S$, estes protocolos têm que considerar a possibilidade dos detectores cometerem falhas. Acreditamos que a utilização de serviços de detecção de falhas com semânticas mais fortes pode não apenas reduzir as considerações necessárias para garantir a corretude de tais protocolos, como também resultar em ganhos de performance substanciais

para estes.

7.3 Trabalhos Futuros

Como citado anteriormente, este trabalho é parte de um projeto maior, que envolve atualmente mais outro trabalho de dissertação de mestrado, desenvolvido no mesmo laboratório por Brito [13]. Este trabalho tem o objetivo de criar um dispositivo em hardware responsável, dentre outras coisas, por criar um canal de comunicação síncrono, sem fio, para o serviço. Sua principal função, entretanto, será implementar um componente *WatchDog*, responsável por monitorar o comportamento do módulo local do serviço e forçar a falha da máquina quando o módulo não se comportar de forma síncrona, conforme requerido pelos protocolos do serviço. Este trabalho também terá o objetivo de migrar do módulo Linux para o dispositivo em hardware, a maior parte da implementação dos protocolos do serviço, reduzindo mais ainda sua influência sobre o desempenho do sistema e as fontes de incerteza que interferem em sua sincronia.

Versões futuras do Delphus poderiam também parametrizar algumas informações que na implementação atual são fixas, como número máximo de máquinas do domínio, número máximo de entidades monitoráveis e notificáveis, faixa de valores possíveis para tamanho do período, valores das variáveis que compõem uma fatia TDMA, como ϵ e T_{com}, dentre outras. Isto tornaria o serviço mais adaptável aos diferentes sistemas onde poderia ser instalado.

Outro trabalho futuro possível é a extensão da funcionalidade de notificação de eventos do serviço para outras classes de eventos. Para isso, as mensagens *heartbeat* poderiam ser alteradas para transmitir mais informações sobre as entidades monitoráveis, como por exemplo o estado dos processos no sistema operacional, a quantidade de memória em uso nas máquinas e o percentual de ocupação da CPU pelos processos. Essas informações poderiam ser usadas para produzir novas classes de eventos, como eventos de processos, e eventos de máquina. Desta forma, entidades poderiam ser notificadas quando um determinado limite de memória em uso fosse alcançado em uma máquina, ou quando um processo tivesse seu estado mudado para bloqueado, por exemplo.

O envio periódico de mensagens de sincronização poderia ser usado futuramente como a

base para a implementação de um serviço de sincronização de relógios para as máquinas da rede. Este seria um benefício a mais da adoção do serviço no sistema.

A ferramenta Olen também poderia ser aperfeiçoada em um trabalho futuro. Ela poderia evoluir para se tornar uma ferramenta de gerenciamento de sistemas distribuídos. Esta ferramenta permitiria a um administrador visualizar, monitorar e manipular outras entidades do sistema, como *switches*, *hubs*, aplicações distribuídas, serviços de rede e sub-redes.

Poderia também haver um trabalho para criar uma implementação do serviço para outros sistemas operacionais como o Unix e o Windows. Isso permitiria que o sistemas com plataformas heterogêneas, pudessem utilizar o serviço e aumentaria a portabilidade das aplicações desenvolvidas para utilizá-lo.

Um dos trabalhos futuros a serem realizados com o Delphus já começa a tomar forma no Laboratório de Sistemas Distribuídos da Universidade Federal de Campina Grande: a implementação de uma versão tolerante a falhas do servidor de aplicações JBoss baseada em replicação passiva. Costa e Brasileiro desenvolveram o JBossFT [20], uma versão de um dos mais populares servidores de aplicações J2EE, o JBoss, que implementa mecanismos de tolerância a falhas. Estes mecanismos baseiam-se no uso de replicação ativa dos serviços do JBoss, apoiada sobre a ferramenta de comunicação em grupo JavaGroup [9]. O projeto em andamento pretende aprimorar o JBossFT para que este possa também adotar a replicação passiva dos seus serviços. O Delphus será utilizado como o serviço de detecção de falhas que permitirá o sistema detectar a falha de seu servidor ativo e assim ativar um servidor passivo para assumir os serviços do JBossFT.

Todavia, o principal trabalho futuro identificado é a análise do possível ganho de desempenho de protocolos de consenso apoiados sobre serviços de detecção com semântica perfeita como o Delphus. Os resultados desta análise poderiam motivar um aumento do uso de detectores com semântica perfeita pelas aplicações distribuídas, mesmo em situações onde um $\diamond S$ pudesse ser usado.

Referências

[1] Design CPN Performance Tool Manual, version 1.0. CPN Group - Department of Computer Science - University of Aarhus, 1999. http://www.daimi.au.dk/designCPN/man/.

[2] Java Code Conventions. Sun Microsystems, 1999. http://java.sun.com/docs/codeconv.

[3] Fault Tolerant CORBA Specification. Object Management Group, 2000. http://www.omg.org.

[4] Linux Kernel Coding Style. Linus Torvalds, 2001. http://www.purists.org/linux.

[5] JUnit, Testing Resources for Extreme Programming. JUnit Project, 2003. http://www.junit.org.

[6] Time Division Multiple Access (TDMA). IEC - International Engeneering Consortium, 2003. http://www.iec.org/online/tutorials/tdma.

[7] Aguilera, M. K., Chen, W., and Toueg, S. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *Procedings of 11th Workshop on Distributed Algorithms (WDAG'1997)* (Saarbrucken, Germany, Sep 1997), pp. 126–140.

[8] Aguilera, M. K., Chen, W., and Toueg, S. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science 200*, 1 (Jun 1999), pp. 3–30.

[9] Ban, B. JavaGroups - Group Communication Patterns in Java, 1998. http://www.cs.cornell.edu/home/bba/Patterns.ps.gz.

[10] Batalha, M. S. G. Serviço CORBA de Diagnóstico de Falhas (SDF). Dissertação de Mestrado, COPIN – Universidade Federal da Paraíba, Campina Grande, outubro 2001.

[11] Bovet, D. P., and Cesati, M. *Understanding the Linux Kernel.* O' Reilly, 2002, ch. Processes. pp. 75.

[12] Bovet, D. P., and Cesati, M. *Understanding the Linux Kernel.* O' Reilly, 2002, ch. Kernel Synchronization. pp. 161–192.

[13] Brito, A. E. M. Projeto e Implementação de um Módulo Síncrono para a Implementação de um Serviço de Detecção de Falhas com Semântica Perfeita para Redes Locais. Proposta de Dissertação de Mestrado - COPIN/DSC/UFCG, 2003.

[14] Brito, A. E. M., de Oliveira, E. W., and Brasileiro, F. V. Delphus: Uma Ferramenta de Detecção de Falhas para Redes Locais. In *Anais do II Salão de Ferramentas do Simpósio Brasileiro de Redes de Computadores* (Natal, RN, Maio 2003).

[15] Casimiro, A., Martins, P., and Veríssimo, P. How to Build a Timely Computing Base using Real-Time Linux. In *Proceedings of the 2000 IEEE International Workshop on Factory Communication Systems* (Porto, Portugal, Sept. 2000), IEEE Industrial Electronics Society, pp. 127–1343.

[16] Chandra, T., and Toueg, S. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM 43*, 2 (Mar 1996), pp. 225–267.

[17] Chandra, T. D., Hadzilacos, V., and Toueg, S. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM 43*, 4 (Jul 1996), pp. 685–722.

[18] Chen, W., Toueg, S., and Aguilera, M. K. On the Quality of Service of Failure Detectors. In *International Conference on Dependable Systems and Networks (DSN'2000)* (New York, USA, Jun 2000), pp. 191–200.

[19] Chungm, P., Huang, Y., Yajnik, S., Liang, D., and Shih, J. DOORS: Providing Fault Tolerance to CORBA Objects. In *The IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 98')* (The Lake District, England, 1998). Poster session.

[20] Costa, A. A., and Brasileiro, F. V. JBossFT: Tolerância a Falhas para Aplicações J2EE. In *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos* (Natal, RN, Maio 2003).

[21] Cristian, F., and Fetzer, C. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems 10*, 6 (Jun 1999), pp. 642–657.

[22] D.Dolev, Dwork, C., and Stockmeyer, L. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM 34*, 1 (1987), pp. 77–97.

[23] de Oliveira, E. W. A. Monitor – Serviços de Diagnóstico de Falhas e Gerenciamento de Aplicações Distribuídas. Monografia do curso de especialização, LaSiD – Universidade Federal da Bahia, Salvador, novembro 2001.

[24] Doudou, A., Garbinato, B., Guerraoui, R., and Schiper, A. Muteness Failure Detectors: Specification and Implementation. In *Proceedings of the* 3rd *European Dependable Computing Conference (EDCC-3), LNCS 1667* (Prague, Czech Republic, Sep 1999).

[25] Dwork, C., Lynch, N. A., and Stockmeyer, L. Consensus in the Presense of Partial Synchrony. *Journal of the ACM 35*, 2 (1988), pp. 288–323.

[26] Ezhilchelvan, P. D. Design and Development of Algorithms for Fault Tolerant Distributed Systems. In *Thecnical Report Series, No. 304* (Newcastle upon Tyne, UK, 1990), Computing Laboratory, University of Newcastle upon Tyne, pp. 15–20.

[27] Felber, P., Guerraoui, R., Défago, X., and Oser, P. Failure Detector as First Class Objects. In *International Symposium on Distributed Objects and Applications* (L'Aquila, Italy, Sep 1999).

[28] Fetzer, C. Enforcing Perfect Failure Detection. In *Proceedings of the 21st International Conference on Distributed Systems* (Phoenix, AZ, April 2001).

[29] Fetzer, C., and Cristian, F. A Fail-Aware Datagram Service. *IEE Proceedings - Software Engineering* (April 1999), pp. 58–74.

[30] Fischer, M. J., Lynch, N. A., and Paterson, M. D. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM 32*, 2 (Apr 1985), pp. 374–382.

[31] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns - Elements of Reausable Object-Oriented Software.* Addison-Wesley, 2000.

[32] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns - Elements of Reausable Object-Oriented Software*. Addison-Wesley, 2000, ch. Singleton. pp. 293-303.

[33] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns - Elements of Reausable Object-Oriented Software*. Addison-Wesley, 2000, ch. Abstract Factory. pp. 87-95.

[34] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns - Elements of Reausable Object-Oriented Software*. Addison-Wesley, 2000, ch. Singleton. pp. 127-134.

[35] Hester, K. Java signals. http://www.geeksville.com/ kevinh/projects/javasignals/, 1999.

[36] Maffeis, S. Piranha: A CORBA Tool for High Availability. *Computer 30*, 4 (Apr 1997), pp. 56–66.

[37] Mullender, S. *Distributed Systems*. Addison-Wesley, 1993, ch. What Good are Models and What Models are Good? pp. 17-26.

[38] Mullender, S. *Distributed Systems*. Addison-Wesley, 1993, ch. Scheduling. pp. 491-509.

[39] Natarajan, B., Gokhale, A. S., Yajnik, S., and Schmidt, D. C. DOORS: Towards High-Performance Fault Tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA2000)* (Antwerp, Belgium, Sept. 2000), pp. 39–48.

[40] Sampaio, L. M., Brito, A. E. M., and de Oliveira, E. W. Detectores de Falhas para Sistemas Assíncronos. In *Anais do Simpósio Brasileiro de Redes de Computadores, Workshop de Tolerância a Falhas* (Natal, RN, Maio 2003).

[41] Sergent, N., Défago, X., and Schiper, A. Failure Detectors: Implementation Issues and Impact on Consensus Performance. In *EPFL, Technical Report SSC* (École Polytechnique Fédérale de Lausanne, Switzerland, 1999).

[42] Tanenbaum, A. S. *Redes de Computadores*. Editora Campus, 1996, ch. A Subcamada de Acesso ao Meio. pp. 278–279.

[43] Tanenbaum, A. S. *Redes de Computadores*. Editora Campus, 1996, ch. A Camada Física. pp. 107.

[44] Tanenbaum, A. S. *Redes de Computadores*. Editora Campus, 1996, ch. Introdução. pp. 9.

[45] Veríssimo, P., and Casimiro, A. The Timely Computing Base Model and Architecture. In *Transactions on Computers – Special Issue on Asynchronous Real-Time Systems 51, 8* (Aug 2002).