UFSM

TRABALHO DE GRADUAÇÃO

AGREGAÇÃO DE FUNCIONALIDADES AO AFDSERVICE

Aluno:
Augusto Ordobás Bortolás

Orientador:
Raul Ceretta Nunes

Santa Maria, RS, Brasil
2004

AGREGAÇÃO DE FUNCIONALIDADES AO AFDSERVICE

Por

Augusto Ordobás Bortolás

Trabalho de Graduação apresentado ao Curso de Graduação em Ciência da Computação – Bacharelado, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de

Bacharel em Ciência da Computação

Curso de Ciência da Computação

Trabalho de Graduação n° 182

Santa Maria, RS, Brasil

2004

Universidade Federal de Santa Maria

Centro de Tecnologia

Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada, aprova o Trabalho de Graduação

AGREGAÇÃO DE FUNCIONALIDADES AO AFDSERVICE

elaborado por

Augusto Ordobás Bortolás

como requisito parcial para obtenção do grau de

Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA

_____

Raul Ceretta Nunes

(Orientador)

_____

Márcia Pasin

_____

Iara Augustin

Santa Maria, dezembro de 2004

“Dedicado a todas as pessoas que confiaram e acreditaram no meu trabalho.”

Agradecimentos

Agradeço a todas as pessoas que confiaram no meu trabalho, e que, principalmente, proporcionaram meios para que eu pudesse executá-lo sem maiores preocupações:

meus pais, Margarita e Bortolás, que me “agüentaram” durante tanto tempo e sempre deram apoio, com muito carinho, tanto nos bons quanto nos maus momentos;

meus irmãos, Natália e Daniel, que sempre me deram apoio quando precisei;

meu orientador, Raul, que sempre esteve à disposição, com muito boa vontade, para resolver problemas e oferecer sugestões e conselhos, muitas vezes no limite do *deadline*;

a todos os colegas de trabalho, principalmente o Rogério, que sempre se colocaram à disposição para ajudar;

a todos os amigos que sempre estiveram presentes, e cujo auxílio psicológico foi fundamental para a realização deste trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

FD	Módulo de detecção de defeitos do AFDService
HTTP	*Hyper Text Transfer Protocol*
HTML	*Hyper Text Markup Language*
IDE	*Integrated Development Environment*
IP	*Internet Protocol*
JSP	*Java Server Pages*
LAN	*Local Area Network*
TCP	*Transmission Control Protocol*
TS	Módulo de predição de *timeout* do AFDService
UDP	*User Datagram Protocol*
WAN	*Wide Area Network*
WWW	*World-Wide Web*

LISTA DE FIGURAS

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Centro de Tecnologia
Universidade Federal de Santa Maria

PROTOCOLOS DE DETECÇÃO DE DEFEITOS:
ESTUDO E IMPLEMENTAÇÃO

Aluno:
Augusto Ordobás Bortolás

Orientador:
Raul Ceretta Nunes

Sistemas Distribuídos Tolerantes a Falhas são cada vez mais requisitados e utilizados por vários tipos de empresas e órgãos governamentais. Para que a tolerância a falhas seja alcançada nestes sistemas é necessário que exista redundância de componentes, sejam eles de *hardware* ou de *software*. Entretanto, para que os dados replicados permaneçam consistentes, usam-se protocolos de gerência de réplicas, os quais realizam algum tipo de acordo entre os componentes. O problema é que o acordo não pode ser resolvido deterministicamente em ambientes assíncronos sujeito a defeitos, como a Internet, devido à característica aleatória do atraso de comunicação (que é desconhecido). A abstração de detectores de defeitos não confiáveis encapsula o problema do indeterminismo dos ambientes assíncronos.

Na prática, quando um serviço de detecção de defeitos executa em ambientes como a Internet, onde existem barreiras (*firewalls*) na sua comunicação, portas de comunicação costumam ser liberadas para que as instâncias do serviço (objetos monitores e monitoráveis) possam se comunicar, gerando falhas na segurança da rede protegida pelo *firewall*.

Este trabalho apresenta a especificação e implementação de funcionalidades no AFDService, um Serviço de Detecção de Defeitos Adaptativo implementado em Java. Estas funcionalidades consistem na implementação de uma solução para o problema de segurança, na agregação de dois novos protocolos de detecção (*push* e *dual*), em um protocolo de acordo, bem como o padrão de projeto *singleton* ao AFDService. Por fim, é exposto como foi realizado o empacotamento da aplicação, característica que torna trivial a instalação e uso do serviço. Também é apresentada a documentação do serviço, como páginas WEB.

Com os resultados obtidos até o momento, demonstra-se que estas contribuições tornam o serviço de detecção mais versátil.

1. INTRODUÇÃO

Segundo Tanenbaum e Steen [TANENBAUM; STEEN, 2002], um sistema distribuído consiste em um conjunto de computadores independentes uns dos outros que parecem aos seus usuários como sendo um único e coerente sistema. Sistemas distribuídos robustos, confiáveis e com alta disponibilidade, executando sobre a Internet, são cada vez mais requisitados, seja por grandes corporações, por empresas de médio e grande porte ou por universidades que desejam uma maior distribuição de suas aplicações. Para que importantes características como robustez, confiabilidade e alta disponibilidade sejam alcançadas, é necessário que exista tolerância a falhas nestes sistemas. A tolerância a falhas é alcançada através da *redundância* de componentes.

Num sistema distribuído, onde os nós podem ser considerados independentes do ponto de vista de falhas, a distribuição tem sido aproveitada para prover tolerância a falhas. Entretanto, a distribuição do estado da aplicação e a característica não confiável da comunicação entre os nós do sistema motivam a exploração de técnicas de tolerância a falhas para sistemas distribuídos. Devido a esta característica não confiável do sistema distribuído, defeitos devem ser descobertos e tratados adequadamente, de modo que a aplicação possa executar normalmente, como se nada tivesse ocorrido. A percepção de defeitos[1] é uma ação explorada por um detector de defeitos, normalmente implementado como um algoritmo distribuído, baseado em troca de mensagens.

De acordo com a percepção de um defeito, pode-se classificá-lo como [CRISTIAN, 1991]:

- defeito de **omissão**: um componente do sistema não responde a uma dada entrada;
- defeito de **resposta**: o componente responde, mas de forma incorreta, através de um valor de saída incorreto (representa um defeito de valor) ou de uma transição

1 No contexto deste trabalho, um defeito corresponde à incapacidade de algum componente (de *software* ou *hardware*) de executar sua função.

de estado que acontece incorretamente (defeito de transição de estado);

- defeito do tipo **colapso** (*crash*): ocorre quando, após a primeira omissão de resposta, o sistema deixa de produzir respostas para as entradas subseqüentes até que seja reinicializado;
- defeito de **temporização**: quando a resposta do componente é funcionalmente correta mas fora do intervalo de tempo especificado. Este tipo de defeito pode tanto ocorrer na forma de uma resposta antecipada como através de uma resposta tardia (defeito de desempenho).

Segundo Jalote [JALOTE, 1994], na hierarquia o defeito de colapso é englobado pelo de omissão, que por sua vez é englobado pelo de temporização e todos eles estão contidos no conjunto mais universal de comportamento arbitrário. O conceito de "englobar" significa que detectores de defeitos desenvolvidos para a classe mais geral também têm a capacidade de detectar defeitos de suas subclasses. Os defeitos de resposta, ou de *computação incorreta*, são englobados pelo modelo arbitrário, portanto são ortogonais aos demais.

Neste trabalho é considerado que a troca de mensagens está sujeita a defeitos de temporização. Apesar de tornar a validação dos algoritmos distribuídos mais complexa, são obtidas garantias reais de funcionamento.

Um fator crítico na segurança dos sistemas distribuídos é o protocolo usado na comunicação entre os nós do sistema. O protocolo UDP é normalmente usado para esta tarefa, quando os processos do sistema distribuído estão hospedados em máquinas em uma mesma rede. No entanto, quando é necessária a comunicação entre máquinas hospedadas em redes distintas, portas de comunicação devem ser liberadas nos *firewalls* das redes, o que introduz problemas na segurança das redes, pois *hackers* podem invadir o sistema facilmente. Protocolos como HTTP podem ser usados como alternativa de comunicação.

Este trabalho propõe a agregação de funcionalidades ao AFDService [NUNES; JANSCH-PÔRTO, 2003], um Serviço de Detecção de Defeitos Adaptativo implementado em Java. Estas funcionalidades consistem na implementação de uma solução para o problema de segurança do AFDService, na agregação de dois novos

protocolos de detecção (*push* e *dual*) e de um protocolo de acordo, na adoção do padrão de projeto *singleton*, no empacotamento da aplicação, característica que torna trivial a instalação e uso do serviço, e na documentação do serviço, que é apresentada como páginas WEB.

Nas seções a seguir serão apresentados os seguintes temas: uma revisão sobre sistemas distribuídos e detectores de defeitos (1.1); uma visão geral sobre o AFDService (1.2); as necessidades do AFDService (1.3); os objetivos do trabalho (1.4); e, finalmente, a forma como este volume está organizado (1.5).

1.1 Sistemas Distribuídos e Detectores de Defeitos

Nas aplicações distribuídas, o modelo de comunicação mais usado entre os processos é o de troca de mensagens [TANENBAUM; STEEN, 2002]. Neste modelo, os processos executam algoritmos que se sincronizam uns com os outros exclusivamente por troca de mensagens através de canais de comunicação. Se este modelo não considerar que a comunicação entre os seus processos pode sofrer defeitos, os algoritmos podem facilmente ser especificados e implementados. Entretanto, como defeitos na comunicação são comuns na Internet, podendo levar a aplicação a um colapso, defeitos devem ser considerados na modelagem do sistema.

Um sistema distribuído tolerante a falhas possui como principal característica redundância de componentes, sejam estes de *software* ou de *hardware*. No entanto, para que os dados replicados permaneçam consistentes, os protocolos que gerenciam as réplicas necessitam de algum tipo de acordo entre os componentes [GUERRAOUI; SCHIPER, 1997]. Porém, o acordo entre os componentes não é passível de ser resolvido deterministicamente num sistema assíncrono sujeito a defeitos [FISCHER; LYNCH; PATERSON, 1985].

Neste contexto, o conceito de detectores de defeitos não confiáveis[2] introduzido por Chandra e Toueg [CHANDRA; TOUEG, 1996], tornou-se uma

2 Estes detectores são chamados não confiáveis pois podem suspeitar erroneamente que um processo correto está falho.

importante abstração para a construção de sistemas distribuídos tolerantes a falhas. Uma de suas utilidades é auxiliar os protocolos de acordo. A abstração de detectores de defeitos não confiáveis encapsula o indeterminismo no atraso de comunicação entre dois nodos distribuídos. Normalmente, os detectores são implementados com o uso de *timeouts*, ou limites de tempo de espera. Mecanismos de detecção de defeitos são particularmente valiosos para aplicações como gerenciamento de transações, replicação, balanceamento de carga, coleta de lixo, e aplicações de monitoramento como supervisão de sistemas de controle [FELBER; DÉFAGO; GUERRAOUI, 1999b].

Uma modelagem para detectores de defeitos, sugerida por Felber et al. (1999a), é apresentada a seguir.

Tipos de objetos utilizados no serviço de detecção/monitoramento:

- **monitores ou detectores de defeitos** – objetos que guardam informações sobre defeitos nos componentes da aplicação distribuída;
- **monitoráveis** – estes objetos são monitorados pelo detector de defeitos;
- **notificáveis** – são objetos registrados pelo serviço de monitoramento para serem notificados, de forma assíncrona, sobre os defeitos do sistema.

Os estados do objeto monitorado, sob o ponto de vista do monitor, são os seguintes:

- **SUSPECTED** – objeto suspeito de falhas;
- **ALIVE** – objeto é considerado não falho;
- **DONT_KNOW** – objeto não está sendo monitorado.

Ainda de acordo com Felber, o modelo de detector de defeitos é classificado, quanto ao fluxo de informações e controle, como:

- ***push*** – neste modelo, os objetos monitorados são ativos, pois periodicamente enviam mensagens de *heartbeat* aos objetos monitores, com o objetivo de informar que não estão falhos;

- ***pull*** – neste modelo, os objetos monitoráveis são passivos. Os objetos monitores enviam periodicamente mensagens de solicitação de estado (*liveness request*) aos objetos monitorados. Caso o objeto monitorado não responda em um determinado *timeout*, ele é considerado suspeito de ter falhado.

Uma alternativa, mais flexível, é o modelo **misto** (*dual*). O objetivo deste modelo é unir as características dos modelos *push* e *pull*, de forma que o detector de defeitos possa interagir com os objetos monitorados usando as duas formas de acesso.

Apesar das vantagens de um modelo sobre outro, não se pode dizer qual deles é o melhor, pois suas peculiaridades podem ser exploradas de acordo com o ambiente onde são utilizados.

Neste trabalho foram implementados os algoritmos *push* e *dual*.

1.2 AFDService – Um Serviço de Detecção de Defeitos Adaptativo

Como exposto, este trabalho visa apresentar a agregação de funcionalidades ao AFDService [NUNES; JANSCH-PÔRTO, 2003], um Serviço de Detecção de Defeitos Adaptativo implementado na linguagem de programação Java [SUN, 2004] [DEITEL, 2001], baseado em tempo máximo de espera (*timeouts*). O AFDService é um serviço que tem como principal função monitorar nodos de um sistema distribuído, e pode ser configurado para adaptar seu *timeout* dinamicamente de acordo com o comportamento do atraso de comunicação. Sua estrutura é formada por dois módulos, de predição (responsável pela adaptabilidade do *timeout* do serviço) e de detecção (responsável pela detecção de defeitos). Ele inicialmente foi projetado para suportar defeitos dos tipos colapso, omissão e temporização. Seu paradigma de comunicação é por troca de mensagens, utilizando-se *sockets* UDP. Neste trabalho, a implementação base do AFDService foi utilizada, e foram agregadas novas funcionalidades.

1.3 Necessidades do AFDService

Ao se implementar algoritmos de detecção de defeitos em uma camada *middleware* sobre redes, como a Internet, facilmente percebe-se um empecilho que dificulta a comunicação entre nós hospedados em redes locais distintas: a presença de *firewalls*.

Para ocorrer a comunicação entre os nós, é necessário que ocorra a liberação de portas de comunicação nos *firewalls* presentes. Isto pode gerar falhas de segurança em ambas as redes, pois um *hacker* pode facilmente descobrir estas brechas na segurança e, também facilmente, pode invadir através delas. Após ter acesso total ao *firewall*, que normalmente é o ponto de acesso da rede interna à Internet, um usuário mal intencionado pode invadir todas as outras máquinas da rede [BORTOLAS; KREUTZ; NUNES, 2004].

Isto é uma situação extremamente perigosa em termos de segurança, pois informações confidenciais armazenadas na rede podem ser acessadas por usuários sem permissão para isto. Uma alternativa para solucionar este problema é o uso do protocolo HTTP, através de servidores WEB, na comunicação entre domínios distintos, e *scripts* JSP [DUANE; MARK, 2000] na comunicação interna da rede, entre o servidor WEB intermediário e a máquina a ser monitorada. Com esta solução não é necessária a liberação extra de portas de comunicação nos *firewalls*, pois normalmente uma porta padrão (80) é utilizada para a comunicação com servidores WEB.

A implementação inicial do AFDService não apresentava solução para o problema da falha de segurança gerada nos *firewalls*. Por isso, uma das metas deste trabalho é apresentar o estudo e implementação de um módulo (adotando a especificação da solução via HTTP) que, agregado ao serviço de detecção, elimine as brechas de segurança citadas anteriormente. Além disso, o serviço necessitava de novas funcionalidades, para oferecer mais opções aos clientes do serviço.

Devido à alta heterogeneidade dos sistemas atualmente, uma maior gama de opções é necessária, para que torne o serviço mais adaptável aos diferentes ambientes de execução. Entre estas heterogeneidades, encontram-se as diferentes topologias de redes, possuindo cada uma delas suas peculiaridades. O AFDService, com novas funcionalidades, possui maior chance de explorar estas características.

O AFDService também necessitava um empacotamento adequado, de forma que pudesse ser instalado e utilizado de forma prática, bem como de um manual, explanando o serviço, que foi realizado neste trabalho.

1.4 Objetivos

Este trabalho possui como objetivo geral apresentar o estudo e implementação de algoritmos de detecção de defeitos, e agregá-los ao AFDService. Os objetivos específicos são os seguintes:

- estudo e implementação de um módulo que permita as instâncias do AFDService se comunicarem via protocolo HTTP em redes locais remotas;
- estudo e implementação de dois protocolos de detecção de defeitos, *push* e *dual*, e a agregação destes ao AFDService, para que os futuros usuários possam testar as várias opções disponibilizadas pelo serviço e adaptá-las aos seus ambientes específicos;
- estudo e implementação de uma camada de consenso no AFDService, para permitir que as instâncias do serviço tenham uma visão global única do sistema distribuído, aumentando a qualidade do serviço;
- empacotamento do serviço, para facilitar sua instalação e uso;
- implementação do padrão de projeto *singleton* no AFDService, para permitir a obtenção de apenas uma instância do serviço por *host*, eliminando as chances de ocorrer conflitos de portas de comunicação*;*

- documentação do serviço, contendo manual de instalação e uso, a ser apresentada em páginas WEB, o que proporcionará uma maneira fácil e prática de buscar informações a respeito das funções oferecidas, e também possibilita que novos desenvolvedores compreendam a estrutura do AFDService mais facilmente.

Além disso, este trabalho tem como objetivo um estudo sobre sistemas distribuídos, mais especificamente, sobre detecção de defeitos e protocolos de acordo. Este estudo serve como embasamento teórico para o desenvolvimento do trabalho.

1.5 Organização do texto

O texto está organizado da seguinte maneira: primeiro é apresentado um estudo teórico sobre detecção de defeitos, onde são introduzidos os detectores de defeitos, bem como sua classificação quanto ao fluxo de informações propagadas entre os componentes (capítulo 2).

A seção seguinte (capítulo 3) contém um estudo sobre o AFDService, apresentando as vantagens de ser implementado como um serviço, sua estrutura e interfaces e a forma de integração de detectores no serviço; a seguir (capítulo 4), serão expostos o projeto e a implementação dos protocolos de detecção e acordo, a adoção do padrão de projeto *singleton*, o sistema para transpor *firewalls* e como foi feito o empacotamento do serviço e a documentação; finalmente, serão apresentadas as conclusões (capítulo 5) e bibliografia (capítulo 6).

2 DETECTORES DE DEFEITOS

Esta seção tem por objetivo apresentar conceitos básicos sobre detectores de defeitos, para que as seções seguintes, as quais dizem respeito à parte de implementação do trabalho, possam ser explanadas. Primeiro é apresentada uma definição para detectores (2.1), e a seguir três algoritmos de detecção, *pull*, *push* e *dual* são estudados (2.2). Finalmente, o algoritmo de consenso é explanado (2.3).

2.1 Definição

Um detector de defeitos *FD* é um algoritmo distribuído formado por um conjunto de módulos de detecção de defeitos fd_i , onde cada um destes módulos é anexado a um componente do sistema distribuído sob monitoramento [NUNES; JANSCH-PÔRTO, 2003]. Cada módulo fd_i monitora os estados de um subconjunto de componentes do sistema distribuído e mantém uma lista dos módulos suspeitos de terem falhado. Em um determinado instante de tempo *t*, a aplicação do usuário pode requisitar o estado funcional de um componente ou subgrupo dos componentes monitorados a um módulo fd_i local. Como os módulos de detecção são considerados não confiáveis, ou seja, podem suspeitar erroneamente de um processo, um módulo pode adicionar um processo *p* à lista de suspeitos mesmo se esse processo estiver correto. Caso o módulo, em outro instante, concluir que a suspeita sobre *p* foi um engano, este pode ser retirado da lista de suspeitos. Desta forma, cada módulo fd_i pode retirar e adicionar processos à sua lista de suspeitos, de acordo com sua percepção momentânea. Além disso, como os processos são distribuídos e cada processo tem acesso apenas ao módulo local, em um mesmo instante de tempo, dois módulos podem apresentar listas de suspeitos diferentes.

Para que se mantenha consistência entre as listas de suspeitos, algumas implementações definem que cada módulo de detecção envie sua lista aos demais módulos. Isto auxilia a formação de uma lista unificada.

Uma exigência deste modelo, apresentado por [CHANDRA; TOUEG, 1996], é que os erros de detecção cometidos não influenciem no comportamento dos módulos suspeitos, pois estes devem se comportar de acordo com sua especificação, mesmo sendo considerado suspeito por outros módulos.

De acordo com [CHANDRA; TOUEG, 1996], os detectores de defeitos podem ser especificados de forma abstrata, através de duas propriedades axiomáticas: abrangência (*completeness*) e precisão (*accuracy*). De forma resumida, a propriedade de precisão diz respeito a capacidade do detector de evitar erros. A propriedade abrangência diz respeito a capacidade do detector para suspeitar de processos que tenham realmente falhado.

Muitas questões sobre o desempenho de detectores de defeitos são levantadas. Um detector de defeitos deve atender a exigências quanto ao número de mensagens trocadas entre processos, quanto à latência e à precisão sobre a suspeita de falha de um processo, e quanto à necessidade de conhecimento dos processos do sistema.

Isto se deve ao fato que tais características variam de acordo com as peculiaridades do meio de transmissão de dados, do modelo de falhas considerado e do aumento no número de processos no sistema. Diversas alternativas de implementação foram desenvolvidas para se adequar a certas exigências, de forma que não existe uma proposta melhor que as demais, mas apenas há aquelas que satisfazem melhor as exigências de certos sistemas [ESTEFANEL; JANSCH-PÔRTO, 2000].

2.2 Classificação Quanto ao Fluxo de Mensagens

Felber, Défago e Guerraoui [FELBER; DÉFAGO; GUERRAOUI, 1999b] apresentam uma classificação de detectores de defeitos baseada na forma como o fluxo de informações entre os componentes (detectores, clientes e objetos monitorados) são propagados no sistema. Esta classificação apresenta as características básicas de dois modelos unidirecionais, *push* e *pull*. Além destes dois, é apresentado um modelo misto, chamado *dual*, que agrega características dos dois modelos unidirecionais. Não existe um modelo mais eficiente e preciso que os outros, pois cada um deles se adequa melhor a certos tipos de ambientes. Esta classificação será mostrada a seguir.

2.2.1 Modelo *Push*

No modelo *push* [FELBER; DÉFAGO; GUERRAOUI, 1999b] a direção do fluxo de controle segue a mesma direção do fluxo de informações. Neste modelo, os objetos monitoráveis são ativos e periodicamente enviam mensagens de *heartbeat*, com o objetivo de informar que estão "vivos" (são mensagens do tipo "*I am alive!*"). Se um monitor não receber uma mensagem de *heartbeat* de um objeto monitorado em um intervalo de tempo, ele suspeita deste objeto ter falhado. Este método é eficiente, pois somente mensagens *one-way* são trocadas no sistema, e podem ser implementadas com as facilidades de *multicast* que certas arquiteturas de redes disponibilizam. A figura 2.1 ilustra como o modelo de detecção *push* é usado para monitorar objetos.

Percebe-se que as mensagens trocadas no sentido monitoráveis –> monitor são diferentes das enviadas do monitor para o cliente (mensagens do tipo "*It is alive!*"), pois, normalmente, o monitor somente notifica o cliente quando houve alguma mudança de estado de algum objeto monitorado, enquanto que as mensagens de

heartbeat são enviadas constantemente.

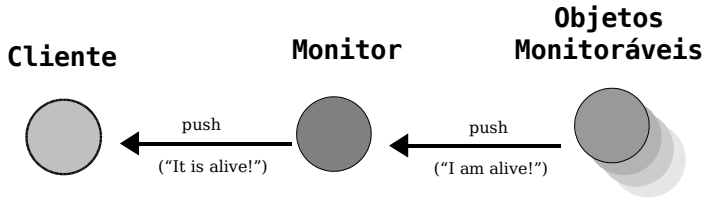

Figura 2.1 – Modelo *push*

O protocolo de troca de mensagens, segundo o modelo *push*, é ilustrado na figura 2.2. O objeto monitorável, periodicamente, envia mensagens "*I am alive"* ao monitor, que, ao receber a mensagem, reinicia o *timeout* daquele objeto, como indica a figura. Caso o *timeout* expire e o monitor não receba mensagens do objeto monitorável, este entra para a lista de suspeitos.

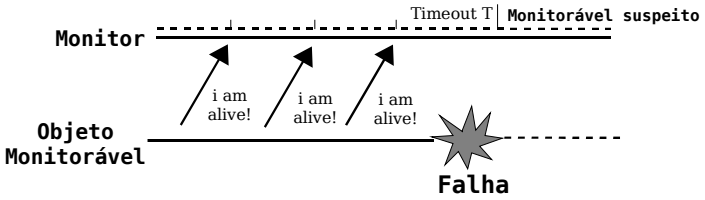

Figura 2.2 – Troca de mensagens no modelo *push*

2.2.2 *Modelo Pull*

No modelo *pull* [FELBER; DÉFAGO; GUERRAOUI, 1999b], a direção do fluxo de informações é oposta à do fluxo de controle, ou seja, os objetos monitoráveis somente enviam mensagens aos monitores quando requisitados pelos mesmos. A figura 2.3 ilustra como o modelo de detecção *pull* é usado para monitorar objetos. Neste modelo, os objetos monitoráveis são passivos. Os monitores periodicamente enviam mensagens do tipo *liveness request* para objetos monitorados. Se um objeto monitorado responder, é sinal de que ele está vivo.

Do ponto de vista do fluxo de mensagens, este modelo é menos eficiente, se comparado ao modelo *push*. Entretanto, a escolha de qual modelo utilizar depende muito da natureza da aplicação.

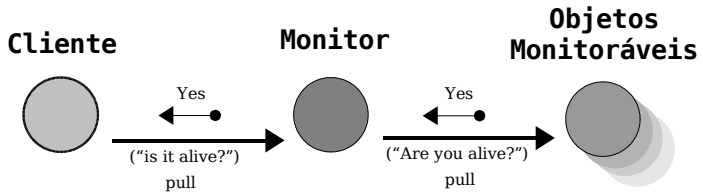

Figura 2.3 – Modelo *pull*

O protocolo de mensagens trocadas no estilo de detecção *pull* é ilustrado na figura 2.4. O monitor envia periodicamente mensagens do tipo "*Are you alive?*" para os objetos monitorados, e espera por uma resposta em um intervalo de tempo. Se não receber uma resposta neste intervalo de tempo, suspeita do processo monitorado. Observe que somente os monitores precisam ter conhecimento sobre os *timeouts* a serem utilizados e ajustados dinamicamente, pois os objetos monitorados são passivos.

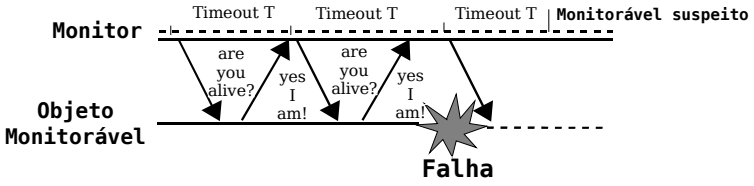

Figura 2.4 – Troca de mensagens no modelo *pull*

2.2.3 Modelo *Dual*

O modelo *dual* [FELBER; DÉFAGO; GUERRAOUI, 1999b] é uma combinação dos modelos *pull* e *push*, onde os dois modelos podem ser usados ao mesmo tempo, com o mesmo conjunto de objetos. O protocolo de troca de mensagens deste modelo funciona da seguinte maneira: o protocolo é dividido em

duas fases distintas. Durante a primeira fase, assume-se que os objetos monitorados utilizam o modelo *push*, portanto, enviam mensagens de *heartbeat* aos monitores.

Quando o *timeout* da primeira fase expirar, os monitores passam para a segunda fase, onde eles assumem que os objetos monitorados que não enviaram mensagens de *heartbeat* durante a primeira fase podem estar falhos; logo, usam o modelo *pull*. Nesta fase, os monitores enviam mensagens de *liveness request* para cada objeto monitorado e esperam por uma mensagem do tipo "*Yes I am!"* (que são semelhantes às do modelo *pull*). Se os objetos monitorados não enviarem mensagens em um determinado período de tempo, eles são considerados suspeitos pelos monitores.

A figura 2.5 ilustra o protocolo de troca de mensagens no estilo *dual*. Neste exemplo, dois objetos são monitorados (M_1 e M_2). O objeto M_1 é monitorado no estilo *push*, enquanto que o objeto M_2 é monitorado no estilo *pull*. O objeto M_1 envia mensagens de *heartbeat* para o monitor, enquanto que M_2 apenas responde mensagens quando requisitado. O monitor utiliza dois *timeouts*, T_1 e T_2, para as fases 1 e 2. Ele espera por *liveness messages* dos objetos monitorados no estilo *push* durante a fase 1. Depois do *timeout* T_1 expirar, o monitor passa para a fase 2 e envia mensagens *liveness request* para os objetos monitorados os quais ele não recebeu mensagens de *liveness request*, e espera durante o *timeout* T_2. Depois do intervalo de tempo T_2, o monitor suspeita de todos os objetos os quais não enviaram mensagens.

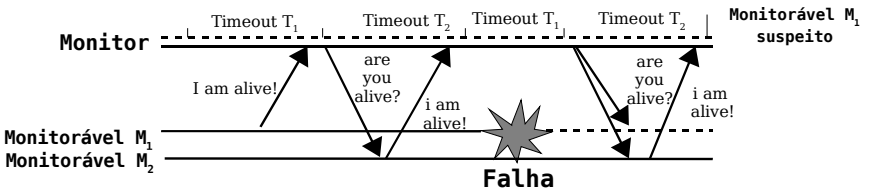

Figura 2.5 – Troca de mensagens no modelo *dual*

No exemplo da figura 2.5, M_1 envia uma mensagem *liveness message*, no intervalo T_1 da fase 1, e a seguir falha. Na fase 1 subseqüente, o monitor não recebe uma mensagem de M_1, portanto, na fase 2 envia uma *liveness request*. Como M_1 não responde, é considerado suspeito pelo monitor.

2.3 Algoritmo de Consenso

Algoritmos de acordo, tais como consenso, *broadcast* atômico ou *commit* atômico, são blocos de construção essenciais para aplicações distribuídas tolerantes a falhas, incluindo aplicações críticas de tempo e transacionais [URBAN; HAYASHIBARA; SCHIPER; KATAYAMA, 2004]. O consenso permite que processos alcancem uma decisão comum sobre algum valor, usualmente binário.

Existem várias abordagens para os algoritmos de consenso, no entanto neste texto será utilizado somente o algoritmo de consenso introduzido por Chandra e Toueg [CHANDRA; TOUEG, 1996] que consiste, basicamente, na realização dos passos ilustrados na figura 2.6:

- primeira fase: os processos enviam suas propostas ao coordenador, pré-estabelecido antes do processo de consenso;
- segunda fase: o coordenador espera um determinado período de tempo, para que a maioria dos processos ((n/2)+1) envie mensagens sugerindo algum valor (esta fase é somente executada pelo coordenador);
- terceira fase: os processos esperam pela decisão do coordenador, e quando a recebem, enviam uma resposta positiva (ACK). Caso o coordenador falhe neste período de tempo, uma mensagem NACK é enviada por cada processo ao coordenador. Neste momento, é iniciada uma nova rodada para a escolha do novo coordenador. A mensagem NACK também pode ser enviada ao coordenador quando os processos não aceitarem o valor decidido pelo algoritmo de consenso;
- quarta fase: o coordenador espera por mensagens da maioria dos processo; se a maioria das mensagens for ACK, o coordenador valida a decisão (esta fase é somente executada pelo coordenador) enviando a todos a decisão final.

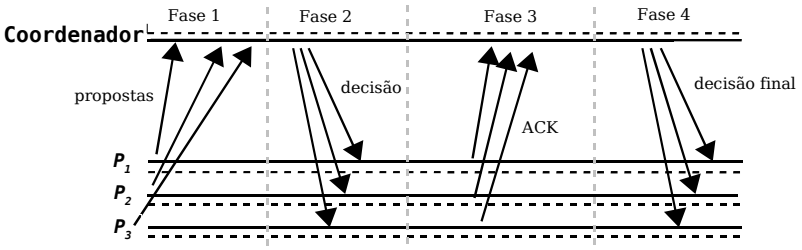

Figura 2.6 – Fases do protocolo de consenso

3 AFDSERVICE

O objetivo deste capítulo é explanar o AFDService [NUNES; JANSCH-PÔRTO, 2003] em mais detalhes para que o leitor compreenda a parte de implementação do trabalho.

As seções a seguir expõem: as vantagens de se ter um detector de defeitos como um serviço (seção 3.1); o projeto e organização do AFDService (seção 3.2); suas interfaces (seção 3.3); a estrutura do módulo FD (seção 3.4); e, por fim, a forma de integração de detectores no AFDService (seção 3.5).

3.1 Detector de Defeitos como um Serviço

Os detectores de defeitos, úteis em vários domínios de aplicações como sistemas de missão crítica e resolução de acordo, podem ser especificados de várias maneiras, algumas das quais são citadas a seguir:

- quanto ao fluxo de informações – *push*, *pull* e *dual* [FELBER; DÉFAGO; GUERRAOUI, 1999b];
- quanto à arquitetura lógica de comunicação – por difusão *(broadcast)* [CHANDRA; TOUEG, 1996], de maneira hierárquica [BRASILEIRO; FIGUEIREDO; SAMPAIO, 2002][FELBER; DÉFAGO; GUERRAOUI, 1999b] ou em anel [LARREA; ARÉVALO; FERNÁNDEZ, 1999];
- quanto ao ajuste do *timeout* – que pode ser fixo ou ajustado de acordo com o comportamento dos atrasos de comunicação na rede.

Além destas características, os detectores podem ser especificados quanto à estratégia de ajuste do *timeout* e quanto às propriedades asseguradas pelo detector [CHANDRA; TOUEG, 1996][LARREA; ARÉVALO; FERNÁNDEZ, 1999]. Devido a esta quantidade de características, os detectores podem ser especificados de várias maneiras. Uma abordagem muito comum é fundir o detector à aplicação cliente, o que aumenta o seu desempenho [SERGENT; DÉFAGO; SCHIPER, 1999]; no entanto, contribui para aumentar a complexidade do projeto, elevando o custo da aplicação. Uma alternativa a esta abordagem, com o intuito de diminuir a complexidade do projeto, é encapsular o detector de defeitos em um serviço, em uma camada *middleware*. Deste modo, o detector pode ser reutilizado para várias aplicações sem a necessidade de seu código ser alterado, e caso ofereça possibilidades de configuração, pode ser adaptado aos mais variados ambientes e aplicações.

3.2 Projeto e Organização

O AFDService é um serviço configurável através de um arquivo de configuração passado ao detector (*ds.cfg*, por padrão, mas outro arquivo pode ser especificado no instanciamento do detector). Sua estrutura é formada por dois módulos: um de detecção (FD) e outro de predição (TS), como ilustra a figura 3.1. A aplicação do usuário interage com o serviço através do módulo FD, o qual configura o módulo TS adequadamente de acordo com as preferências do usuário expressas no arquivo de configuração, ou indicadas dinamicamente durante a execução. Os módulos FD interagem entre si para realizar o monitoramento remoto, através da rede de comunicação.

A monitoração da acessibilidade dos *hosts* remotos é tarefa do módulo de detecção, mas a estimativa do próximo *timeout* a ser utilizado depende da previsão realizada pelo módulo de predição, o qual é informado constantemente sobre os tempos de comunicação observados.

Considera-se que cada *host* possui apenas uma instância do serviço em execução.

Cada módulo possui um repositório de diferentes algoritmos de detecção ou predição. Estes repositórios são controlados por gerentes de estratégias. O *FDManager* é o gerente do módulo FD e o *PredManager* é o gerente do módulo TS. O papel destes gerentes é tornar flexível a escolha das estratégias (de adaptação do *timeout* e detecção de defeitos) a serem utilizadas. Cada módulo FD possui um módulo TS agregado a ele, o qual é responsável pela adaptação dinâmica do *timeout*.

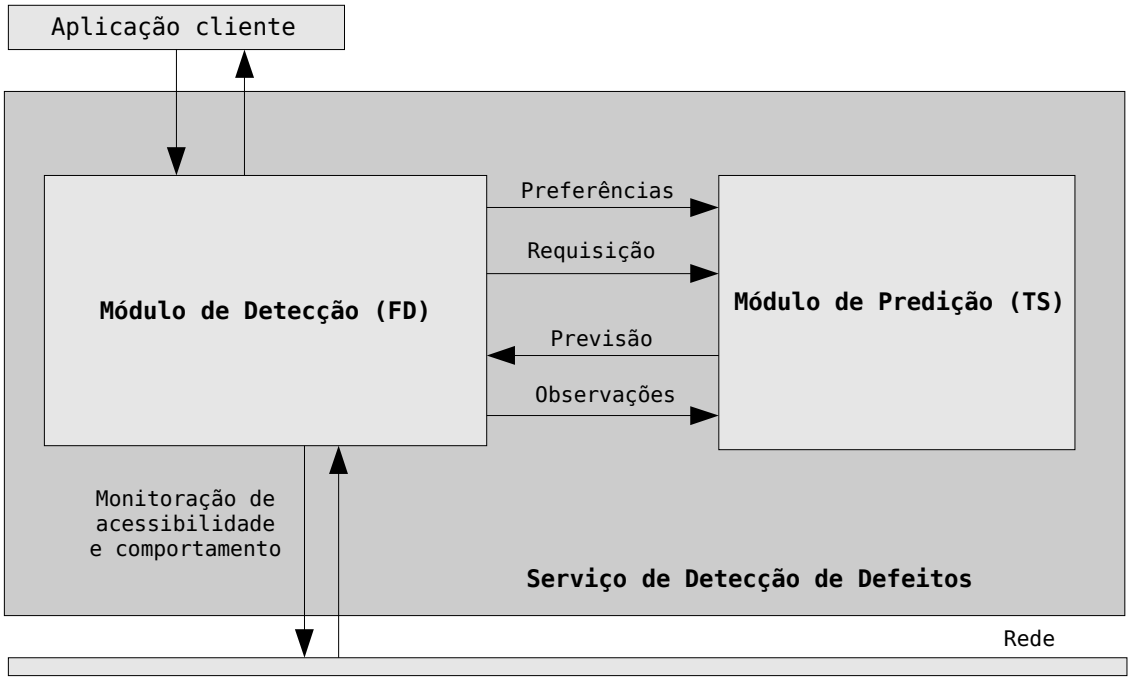

Figura 3.1 – Arquitetura do AFDService

A primeira versão do serviço possui, agregado ao módulo FD, um modelo de detecção de defeitos *pull*, e ao módulo TS, sete estratégias para adaptação dinâmica do *timeout* e três tipos de margem de segurança.

Como este trabalho é relacionado com a implementação de algoritmos distribuídos, o módulo FD foi estudado e utilizado para implementação.

Ambos os módulos (FD e TS) adotam o padrão de projeto *Strategy* [GAMMA; HELM; JOHNSON; VLISSIDES, 1994][PETER, 2004], que permite uma fácil agregação de novos algoritmos de detecção de defeitos (referenciados como “estratégias” no padrão de projeto) ou de novos mecanismos de predição de valores.

3.3 Interfaces

O AFDService possui um conjunto de interfaces baseado na solução proposta por [FELBER; FAYAD; GUERRAOUI, 1999a], portanto possibilita a aplicabilidade de um serviço de detecção em qualquer contexto. A arquitetura de interfaces para um serviço de detecção configurável é ilustrado na figura 3.2, onde o retângulo pontilhado especifica as interfaces orientadas à aplicação.

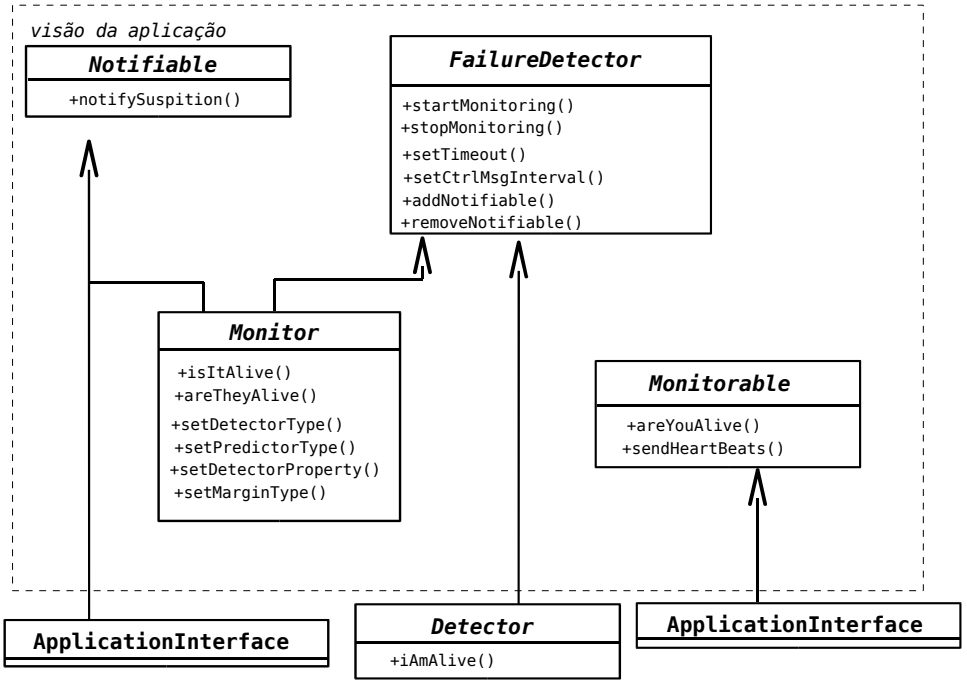

Figura 3.2 – Conjunto de interfaces do AFDService

Basicamente, devem existir 3 componentes principais no ambiente de detecção, como foi citado anteriormente: um monitor (*Monitor*), um notificável (*Notifiable*) e um monitorável (*Monitorable*). O componente monitor irá observar o componente monitorável e irá avisar o componente notificável caso necessário.

Pela figura 3.2, é possível perceber que se torna trivial a utilização do serviço pela aplicação cliente. Para que uma aplicação possa ser notificada a respeito de mudanças no estado dos componentes monitorados, deve implementar a interface *Notifiable*, portanto irá implementar o método *notifySuspition()*, o qual será invocado pelo detector em caso de mudança de estado de algum *host* monitorado. Uma aplicação que deseje ser monitorada, deve implementar a interface *Monitorable*. Novos algoritmos de detecção podem ser adicionados, desde que implementem a interface *Detector*.

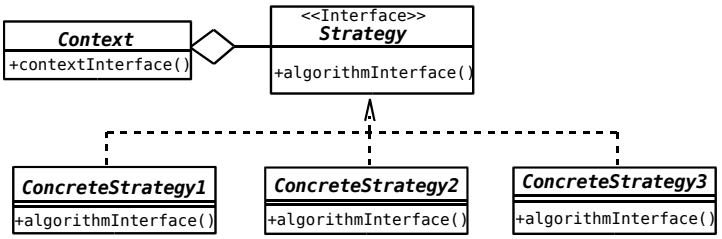

Figura 3.3 – Estrutura do padrão *Strategy*

Ambos os módulos TS e FD adotam o padrão de projeto *Strategy* [PETER, 2004], o qual permite que uma família de algoritmos seja utilizada de modo independente e seletivo. A figura 3.3 ilustra a estrutura do padrão de projeto *Strategy*. Segundo este padrão, temos um *Context*, que representa o gerente dos algoritmos. Os algoritmos correspondem às estratégias concretas (*ConcreteStrategy*), que implementam uma mesma interface (*Strategy*), de modo que o gerente (*FDManager*, na figura 3.4) possa chamar métodos das estratégias concretas sem conhecer sua implementação.

Substituindo as classes do módulo FD no padrão *Strategy*, chega-se à arquitetura apresentada na figura 3.4.

O padrão *Strategy* permite que novos algoritmos sejam inseridos de maneira fácil no serviço, sendo necessário somente a adição da nova classe e referenciando esta classe no gerente de estratégias.

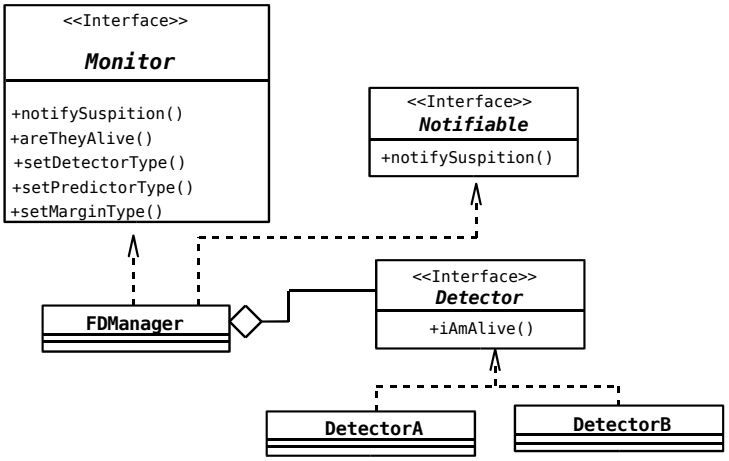

Figura 3.4 – Arquitetura do módulo FD

3.4 Módulo FD

O módulo de detecção de defeitos é implementado como um pacote em Java, nomeado FD. A figura 3.5 ilustra o diagrama de classes do pacote. O núcleo da arquitetura do módulo FD é formado pelas interfaces *Monitor* e *Detector*. Deste modo, o detector da classe *PullDetector* implementa a interface *Detector*, enquanto o gerente das estratégias de detecção, a classe *FDManager*, implementa a interface *Monitor*. Nesta estrutura, novos algoritmos de detecção podem ser adicionados ao módulo, desde que implementem a interface *Detector* e suas identificações sejam incluídas na classe *FDManager*. Para que a aplicação possa escolher, através do método *setDetector()*, uma das estratégias de detecção do repositório, a classe *FDManager* deve conhecer quais são as estratégias de detecção implementadas. A classe *FDManager* corresponde à classe *Context* no padrão *Strategy*.

A idéia de instanciação do AFDService é a seguinte:

No módulo de detecção, a identificação de objetos monitoráveis (nós do sistema distribuído) é realizada por um objeto da classe *Member*, o qual contém os seguintes atributos: nome, endereço IP e uma porta de recepção de mensagens. Estes objetos podem ser adicionados no arquivo de configuração, ou passados diretamente pela aplicação cliente, pelo método *startMonitoring()*. A manutenção do estado destes objetos é realizada pelo *FDManager*, o qual mantém uma lista com os membros suspeitos (*suspectList*).

Mensagens de monitoramento são encapsuladas num objeto serializável do tipo *Message*, o qual tem como atributos: o tipo e o número da mensagem de monitoramento; o endereço IP do emissor e destinatário; e dois campos para informações específicas do protocolo de detecção.

Como exemplo de uso destes, no detector do estilo *pull*, um campo é utilizado para carregar o instante de tempo de envio da mensagem de requisição de estado e o outro campo é usado para informar ao objeto monitorado a porta para onde o reconhecimento deve ser direcionado.

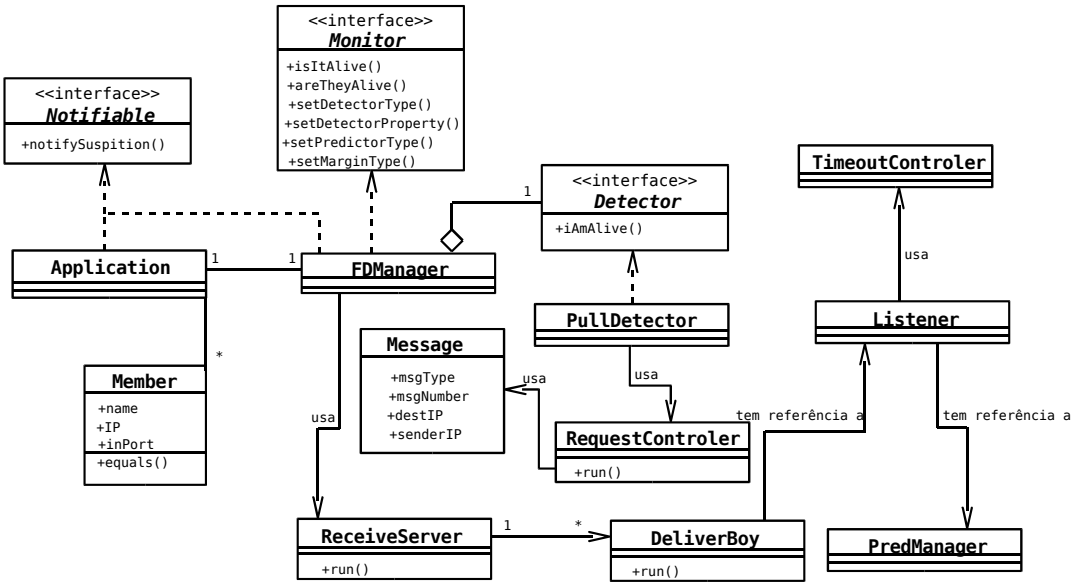

Figura 3.5 – Diagrama de classes do módulo FD

Para comunicação com os módulos de detecção remotos, o AFDService implementa uma classe servidora de recepção de mensagens (*ReceiveServer*), a qual

é responsável por receber as mensagens provenientes de todos os monitores e monitoráveis associados ao módulo local. Esta arquitetura favorece a troca dinâmica da estratégia de detecção, pois ao destruir um dado detector, o ponto de entrada do nó não é destruído. Para resolver o problema de concorrência de acesso à classe receptora de mensagens, foi criada uma *thread* "roteadora" (*DeliverBoy*) [ESTEFANEL, 2000], a qual tem como função tratar cada mensagem recebida. Quando uma mensagem chega ao ponto de entrada, este cria um *DeliverBoy* para entregar a mensagem para o seu devido lugar. A *thread DeliverBoy* é destruída assim que entrega a mensagem. Na arquitetura do AFDService, a *thread DeliverBoy* possui duas funções distintas: serve como um objeto monitorável para outras instâncias do serviço de detecção e como roteador de mensagens para as *threads* de escuta (*listeners*) do detector.

O detector *PullDetector*, primeiro algoritmo de detecção a ser desenvolvido no AFDService, envia mensagens de solicitação de estado, periodicamente, aos objetos monitorados. Estas requisições são feitas através de um objeto do tipo *RequestControler*, o qual é escalonado de tempos em tempos pelo *PullDetector*. Para controlar o *timeout* de cada nó monitorado, o *PullDetector* instancia um objeto de escuta (*Listener*) para cada nó monitorado. Os objetos da classe *Listener* controlam o *timeout* com o auxílio de um objeto da classe *TimeoutControler*, o qual é escalonado para atuar *timeout* instantes de tempo à frente. Caso não seja desescalonado a tempo, coloca o objeto monitorado na lista de suspeitos, invocando o método *notifySuspition ()* da classe *FDManager*.

Ao ser instanciado, um objeto da classe *FDManager* pode receber uma lista de membros a serem monitorados, ou então esta lista pode ser informada através da invocação do método *startMonitoring()*. Se a lista estiver vazia, ou conter apenas a identificação do nó local, o AFDService faz do nó local um objeto monitorado.

Portanto, mensagens que não forem de solicitação de estado são prontamente ignoradas. Se a lista de membros conter algum nó remoto, o AFDService opera localmente como um detector de defeitos sobre aqueles membros monitoráveis, podendo também ser monitorado.

Uma aplicação cliente que deseje ser notificado assincronamente sobre mudanças de estado de objetos monitorados deve implementar a interface *Notifiable*. No entanto, a aplicação cliente pode realizar consultas sincronamente, invocando o método *isItAlive()*, ou *areTheyAlive()*, do objeto da classe *FDManager*.

3.5 Forma de Integração de Detectores no AFDService

Como citado anteriormente, os dois módulos do AFDService adotam o padrão de projeto *Strategy*, tornando a agregação de novos algoritmos uma tarefa fácil. A integração de novos detectores de defeitos no AFDService segue os seguintes passos: a nova classe a ser adicionada deve implementar a interface *Detector*; a classe *FDManager* será modificada para conhecer a partir de então o detector adicionado; como o detector implementa a interface genérica *Detector*, instâncias da classe *FDManager* podem invocar seus métodos sem precisar conhecer sua implementação.

4 PROJETO E IMPLEMENTAÇÃO DAS FUNCIONALIDADES

Esta seção tem como finalidade expor o projeto e a implementação deste trabalho. Primeiro, será apresentado o diagrama de classes modificado do AFDService, contendo as novas classes adicionadas (seção 4.1). A adoção do padrão de projeto *singleton* é explanada na seção 4.2. A seguir, serão apresentadas as implementações dos protocolos *push* e *dual*, nas seções 4.3 e 4.4, respectivamente. A seção 4.5 explana a implementação do sistema para transpor *firewalls*, e a seção 4.6 a implementação do protocolo de consenso. Por fim, é apresentado como foi realizado o empacotamento do serviço (seção 4.7) e as páginas HTML onde foram descritas a documentação e manual de instalação do serviço (seção 4.8).

4.1 Diagrama de Classes Modificado

O diagrama de classes do módulo FD apresentado na seção 3.4 foi modificado, sendo adicionadas novas classes, correspondentes aos dois algoritmos de detecção, *push* (classe *PushDetector*) e *dual* (classe *DualDetector*) e o sistema para transpor *firewalls* (classe *HttpRequestControler*). Estas classes estão ilustradas na figura 4.1.

De acordo com a figura, as classes *PushDetector* e *DualDetector*, assim como a classe já existente *PullDetector*, implementam a interface *Detector*, para que possam ser referenciadas pelo gerente do módulo de detecção, o *FDManager*. Estas duas novas classes possuem referência à classe *RequestControler*, para que possam requisitar e enviar seu estado aos outros detectores.

A classe *RequestControler*, por sua vez, cria uma instância da classe *HttpRequestControler*, caso não consiga se comunicar com algum dos outros detectores de defeitos monitorados, para se comunicarem pelo protocolo HTTP.

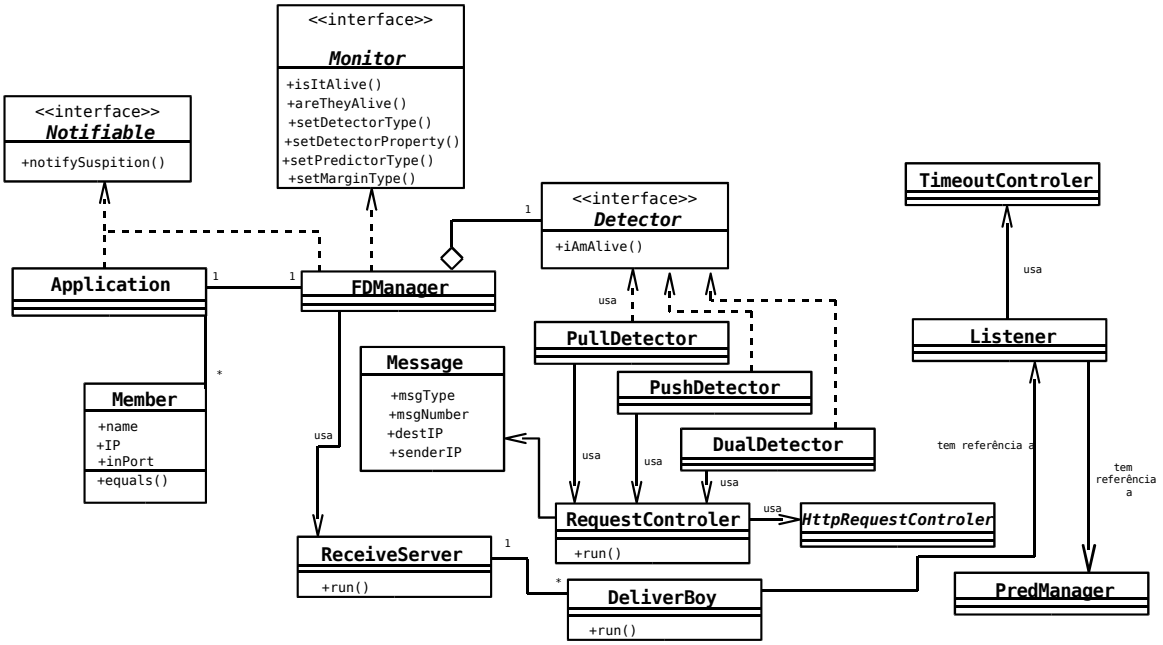

Figura 4.1 – Diagrama de classes do módulo FD modificado

4.2 Padrão de Projeto *Singleton*

A idéia do padrão de projeto *singleton* consiste basicamente em uma forma de forçar o usuário a criar apenas uma instância de uma classe qualquer, prevenindo-o de gerar conflitos entre várias instâncias de classe.

Originalmente, a aplicação cliente do serviço de detecção de defeitos interagia com o mesmo através da classe *FDManager*, criando instâncias do serviço através do construtor da classe, *FDManager()*. O construtor da classe foi substituído pelo método estático da classe *FDManager getInstance()*, que retorna uma única instância do serviço. Caso este método seja invocado várias vezes, ele retornará uma referência da única instância criada. Para a construção do método *getInstance()*, foi adicionado o atributo de classe estático *instance* em *FDManager*, que referencia objetos da própria classe. Quando da primeira invocação do método *getInstance()*, *instance* será *null*, portanto uma nova instância da classe será criada com o construtor padrão, e a referência criada será atribuída à *instance*.

Assim que novas invocações do método *getInstance()* forem feitas, é verificado se *instance* referencia algum objeto. Em caso de resposta positiva, esta referência é retornada pelo método, ao invés de uma nova instância ser criada.

No AFDService, o padrão de projeto *Singleton* eliminou a possibilidade de duas instâncias tentarem acessar as mesmas portas de comunicação na mesma máquina.

4.3 Protocolo *Push*

Basicamente, para a implementação do protocolo *push*, foi criada uma classe no pacote FD, *PushDetector*, cuja principal diferença com relação ao detector *pull* é possuir um vetor contendo os objetos monitores, de modo que possa lhes enviar mensagens "*I am alive!*". Algumas classes foram modificadas (*DeliverBoy*, *Listener*, *RequestControler* e *TimeoutControler*) para suportarem este protocolo.

Como explanado na seção 2, na implementação do protocolo *push* os objetos monitorados devem saber a taxa de envio de mensagens aos seus monitores. Esta tarefa foi resolvida da seguinte maneira: quando da instanciação de um monitor *push*, este envia para seus monitorados uma mensagem de inicialização do tipo *"Push_Init"*, indicando a taxa de envio de mensagens, como ilustrado na figura 4.2. Assim que recebem esta mensagem, os monitorados adicionam o emissor da mensagem em sua lista de monitores e passam a lhes enviar mensagens periodicamente, de acordo com a taxa R indicada na mensagem. Também foi criado o método *addMonitor()*, na classe *PushDetector*, para que os detectores possam adicionar monitores dinamicamente, quando o serviço já está em execução. Este método é invocado quando um objeto monitorado recebe uma mensagem *"Push_Init"*.

Caso o monitor resolva mudar sua taxa de recebimento de mensagens (R), envia uma mensagem do tipo *"Push_Reset_Rate"* aos monitorados.

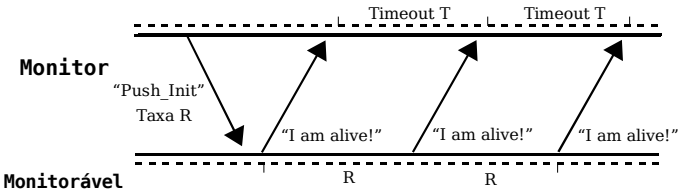

Figura 4.2 – Inicialização dos monitoráveis

Os monitorados irão destruir seus objetos de envio de mensagens (do tipo *RequestControler*), criando um novo objeto de envio, com uma nova taxa de envio, especificada na mensagem, como ilustra a figura 4.3.

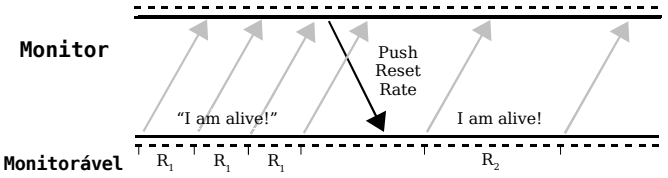

Figura 4.3 – Reset da taxa de envio de mensagens

Modificações nas seguintes classes, já existentes no pacote FD, foram realizadas:

- *FDManager*: o gerente de estratégias foi modificado para que possa reconhecer detectores do estilo *Push*;
- *DeliverBoy*: esta *thread* "roteadora" de mensagens passa a reconhecer mensagens dos tipos "*I am alive!*", "*Push_Init*", "*Push_Reset_Rate*";
- *Listener*: objetos desta classe passam a tratar dos *timeouts* de detectores *push*, recebendo mensagens "*I am alive!*" de objetos *DeliverBoy*;
- *TimeoutControler*: os objetos deste tipo passam a ser escalonados de forma diferente, pois devem executar R segundos de tempo à frente após o recebimento de uma mensagem "*I am alive!*" no *ReceiveServer*. No modelo *pull*, o *TimeoutControler* era escalonado no momento de envio da mensagem "*Are you*

alive?” aos monitorados, de modo que pudesse ser cancelado no momento da resposta.

4.4 Protocolo *Dual*

Para o desenvolvimento deste protocolo, foi criada a classe *DualDetector*, no pacote FD, que monitora detectores do estilo *push* e *pull*. Do mesmo modo que o detector *push*, quando da sua instanciação, o detector dual lê o arquivo de configuração, no qual estão descritos os *hosts* monitoráveis, e lhes envia uma mensagem do tipo *“Push_Init”*. Se o monitorado for *push*, adiciona o monitor em sua lista. Se o monitorado for *pull*, este irá ignorar a mensagem de inicialização, pois sua função é apenas responder mensagens do tipo *“Are you alive?”*. Após o envio da mensagem de inicialização *push*, o monitor espera um *timeout* T_1 (corresponde à fase 1 do protocolo dual especificado na seção 2) pelo recebimento de mensagens dos monitorados *push*. Após este intervalo de tempo, envia mensagens *“Are you alive?”* para os monitorados que não lhe enviaram mensagens na primeira fase, e espera um *timeout* T_2.

Esta questão dos *timeouts* foi resolvida da seguinte forma: como *threads* da classe *TimeoutControler* cuidam de adicionar monitorados na lista de suspeitos, são escalonados $T_1 + T_2$ tempos à frente, após o início do protocolo, sendo que, caso um objeto não envie uma mensagem em ambas as fases, é considerado suspeito. Objetos da classe *RequestControler* possuem como objetivo, nos monitores *dual*, fazer a requisição de estado (nos monitorados *dual*, sua função é enviar mensagens *“I am alive!”*). Na segunda fase do protocolo, são escalonados T_1 tempos à frente, após o início do protocolo, como ilustra a figura 4.4.

Se o *TimeoutControler* não for desescalonado a tempo (caso o monitor não receba alguma mensagem do monitorado), este adiciona o monitorado na lista de suspeitos do detector.

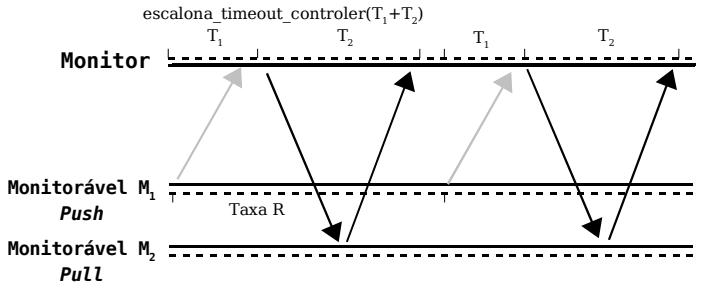

Figura 4.4 – *Timeout* no protocolo dual

Foram feitas modificações nas seguintes classes do pacote FD:

- *FDManager*: o gerente de estratégias foi modificado para reconhecer detectores do estilo *dual*;
- *Listener*: esta classe passa a tratar dos *timeouts* de detectores *dual*;
- *TimeoutControler*: os objetos deste tipo passam a ser escalonados de forma diferente, pois devem executar T_1+T_2 segundos de tempo à frente após o recebimento de mensagens dos monitorados.

4.5 Sistema Para Transpor *Firewalls*

Esta seção tem como objetivo explanar os requisitos para o funcionamento e a implementação do sistema para transpor *firewalls*. A subseção 4.5.1 introduz o sistema, explicando seu funcionamento básico. A seguir, na subseção 4.5.2, são expostos os requisitos e as configurações necessárias para que o sistema possa executar. Por fim, na seção 4.5.3, a implementação do sistema é apresentada.

4.5.1 Funcionamento do Sistema

A proposta de solução para o problema das portas liberadas nos *firewalls* foi a

utilização do protocolo HTTP, ao invés de se usar o método tradicional, por *sockets* UDP. Deste modo, são utilizados servidores WEB como intermediários na comunicação entre as instâncias dos detectores hospedadas em domínios distintos, e *scripts* JSP para comunicação interna da rede, entre o servidor WEB intermediário e a máquina a ser monitorada.

Com esta solução é necessário que alguma porta de comunicação seja liberada no *firewall*, mas normalmente uma porta padrão (80) é utilizada para a comunicação com servidores WEB, e isto não acarreta em maiores problemas de segurança. A figura 4.5 ilustra o processo de monitoramento através do protocolo HTTP.

Pela figura 4.5, é possível perceber que o detector de defeitos monitora uma máquina hospedada em uma rede remota, portanto verifica a acessibilidade do objeto monitorado através de uma requisição HTTP à um servidor WEB, pela porta 80. O servidor WEB irá tratar a requisição (passo 1) e, por meio de um cliente *socket* UDP contido no *script* Java na página JSP requisitada, se comunicará com a máquina monitorada (passo 2), que possui um servidor *socket* UDP. Esta máquina responderá ao servidor socket contido na página JSP (passo 3). O servidor WEB finalmente enviará o resultado da consulta ao objeto monitor (passo 4).

Foi adicionada uma classe no pacote FD, *HttpRequestControler*, que possui como objetivo realizar a comunicação com objetos hospedados em redes protegidas por *firewalls*, de modo que não sejam geradas brechas na segurança. Esta classe faz requisições HTTP à páginas JSP hospedadas em servidores WEB, caso a comunicação por *sockets* UDP falhe.

Deste modo, no caso dos detectores *pull*, a porta do *firewall* deve ser especificada no arquivo de configuração do monitor. Então o monitor irá requisitar o estado dos objetos monitorados através da classe *HttpRequestControler*, ao invés da classe *RequestControler*. Feita a requisição HTTP de uma página JSP no servidor WEB, um *script* Java contido na página JSP irá criar um servidor e um cliente *socket* UDP, para requisitar o estado e esperar uma resposta do objeto monitorado. Após a resposta HTTP, o objeto da classe *HttpRequestControler* irá ler os dados da resposta e enviar estes dados ao *Listener* correspondente ao objeto monitorado.

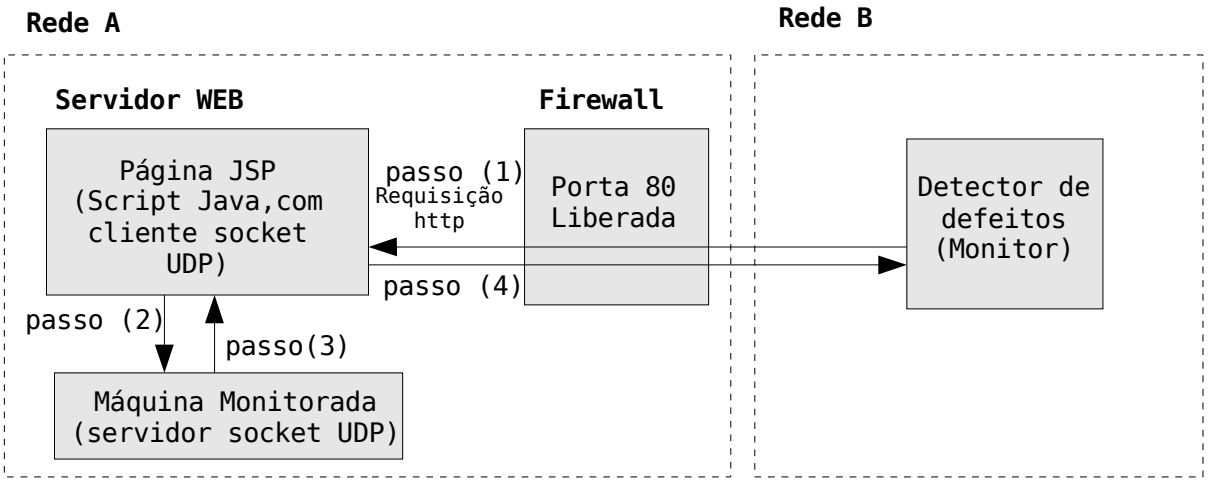

Figura 4.5 – Monitoramento remoto via HTTP

No caso dos detectores *push*, se o monitor não conseguir enviar mensagens de inicialização *"Push_Init"* após três tentativas, utiliza um objeto da classe *HttpRequestControler* para enviar estas mensagens.

Caso a rede hospedeira do monitor também seja protegida por *firewall*, a mensagem de inicialização *"Push_Init"* irá conter a porta de comunicação deste *firewall*, de modo que os monitorados possam, também através da classe *HttpRequestControler*, enviar mensagens *"I am alive!"*.

No protocolo *dual*, ocorre o mesmo sistema descrito nos protocolos *push* e *pull*. As mensagens de inicialização, requisição de estado e envio de estado são geradas pela classe *HttpRequestControler*.

4.5.2 Configuração e Requisitos

Para que o sistema possa executar, é necessário a instalação de um servidor WEB que contenha um contêiner JSP. Para a realização de testes preliminares foi utilizado o servidor WEB Tomcat [APACHE, 2004b].

Também é necessário que uma porta de comunicação seja liberada no *firewall* da rede, e esta porta deve ser disponibilizada ao servidor WEB. A seguir, é necessário que uma aplicação WEB seja criada (esta informação está contida na documentação do servidor WEB utilizado) e que a página JSP utilizada pelo serviço de detecção seja inserida nesta aplição.

Por padrão, é utilizada a URL "*http://IP:80/afdservice/afdservice.jsp*", onde "IP" representa o identificador do *firewall*, "80" é a porta de comunicação, "afdservice" é a aplicação WEB e "afdservice.jsp" é a página que contém o *script* java.

O último requisito é a inserção de entradas no arquivo de configuração do serviço de detecção de defeitos, *ds.config*. Vamos supor que um detector A irá monitorar um detector B, hospedado em uma rede remota. No arquivo de configuração do detector A deve estar contido o IP, a porta de comunicação e o IP do *firewall* do *host* B, de modo que o sistema para transpor *firewalls* seja inteligente, e utilize o protocolo HTTP caso não consiga comunicação por UDP.

4.5.3 Implementação do Sistema

O sistema para transpor *firewalls* foi implementado de forma que somente uma parte do serviço de detecção de defeitos, mais especificamente a classe *RequestControler*, se preocupe com este detalhe.

A invocação dos métodos *sendHeartBeat()* e *sendReqMessage()*, que têm como função enviar mensagens *"I am alive!"* e "*Are you alive?*", respectivamente, é substituída pela invocação dos métodos *sendHTTPHeartBeat()* e *sendHTTPReqMessage()*, caso a comunicação por UDP falhe. Estes métodos estão implementados nas classes de detecção de defeitos *PushDetector*, *PullDetector* e *DualDetector*, mas são invocados pela classe *RequestControler*. A figura 4.6 ilustra o trecho de código que realiza o envio de mensagens "*I am alive!*" pelo protocolo HTTP.

Este trecho de código está presente no método *sendHTTPHeartBeat()*, e representa o passo 1 da figura 4.5.

Suponhamos que um detector A do estilo *push* esteja enviando mensagens *"I am alive! "* para um detector B, situado em uma rede remota. O detector A irá executar o trecho de código ilustrado na figura 4.6. A linha 8 desta figura mostra que o detector irá requisitar uma página JSP a um servidor WEB, passando como parâmetro o próprio IP, para que o detector B saiba quem enviou a mensagem, o tipo da mensagem, para saber o protocolo utilizado, o número de seqüência da mensagem e a porta de entrada UDP do objeto monitor. Caso o detector B queira mudar a taxa de envio de mensagens do monitorado A, irá escrever, no código da página "afdservice.jsp", as *Strings* "ResetRate=yes" e a nova taxa de envio. O monitorado A lê os dados presentes na página HTML retornada (linhas 12 e 14), e invocará o método *resetRate()* (linha 15) passando como parâmetro a nova taxa de envio de mensagens, o que irá mudar a taxa de envio do detector local.

```
1. Member member = member;
2. String webServIP = member.webServerIP.getHostName();
3. String myIP = this.myIP;
4. String num = this.msgNumber;
5. String port = member.inPort;
6. String msgType = "I am alive!";
7. String param = "?myIP="+myIP+"&msgType="+msgType+"&port="+port+"&number="+num;
8. URL url = new URL("http://"+webServIP+":80/afdservice/afdservice.jsp"+param);
9. BufferedReader in;
10.String inputLine;
11.in = new BufferedReader(newInputStreamReader(url.openStream()));
12.inputLine = in.readLine();
13.if(inputLine.equals("ResetRate=yes")) {
14.String rate = in.readLine();
15.this.resetRate(rate);
16.}
```

Figura 4.6 – Envio de mensagem "*I am alive!*"

Analisando o outro lado do sistema, com relação ao servidor WEB e o monitor, tem-se o trecho de código, ilustrado na figura 4.7.

O servidor WEB presente na rede do detector B recebe a requisição da página *afdservice.jsp* do monitorado A, e executa o trecho de código java presente na página JSP, ilustrada na figura 4.7. O servidor lê os dados da requisição (linhas 8 a 12), cria um novo objeto da classe *Member*, que representa o objeto monitor que receberá a mensagem (linha 13) e cria e preenche a mensagem a ser enviada para o monitor (linhas 14 a 18). A mensagem é enviada com o método utilitário *send()*, o qual cria pacotes UDP (mensagem) e os envia ao monitor (linha 20).

Deste modo, o detector B reconhece o estado do monitorado A, e reinicia o seu *timeout*, esperando pela próxima mensagem. Este trecho de código representa o passo 2 da figura 4.5.

Os dois trechos de código apresentados nas figuras 4.6 e 4.7 explanam, basicamente, como funciona o sistema de envio e recepção de mensagens via protocolo HTTP, que é o cerne do sistema para transpor *firewalls*.

```
1. <%@ page import="java.util.*, java.net.*, java.io.*, FD.Utils, FD.* "%>
2. <%! String msgType, myIP, destIP, port, firewall, monitorPort, msgNumber; %>
3. <%! DatagramSocket outMail; %>
4. <%! Member m; %>
5. <%! Message msg; %>
6.
7. <%
8. msgType = request.getParameter("msgType");
9. myIP = request.getParameter("myIP");
10.destIP = request.getParameter("destIP");
11.port = request.getParameter("port");
12.msgNumber = request.getParameter("msgNumber");
13.m = new Member(destIP, Integer.parseInt(port));
14.msg = new Message();
15.msg.number = Integer.parseInt(msgNumber);
16.msg.type = msgType;
17.msg.sender = InetAddress.getByName(myIP);
18.msg.dest = InetAddress.getByName(destIP);
19.DatagramSocket outMail = new DatagramSocket();
20.Utils.send(msg, m, outMail);
21.%>
```

Figura 4.7 – afdservice.jsp

4.6 Protocolo de Consenso

Como citado no capítulo 2, este trabalho implementa o protocolo de consenso de Chandra e Toueg [CHANDRA; TOUEG, 1996], o qual tem o propósito de manter consistência nas listas de objetos suspeitos implementada entre diferentes detectores, aumentando a qualidade do serviço de detecção.

O protocolo foi implementado da seguinte forma: um monitor contém, no seu arquivo de configuração, todos os componentes monitoráveis do sistema. Quando é instanciado, este monitor envia uma mensagem para os componentes do sistema, com os seguintes campos: o IP e a porta de todos os componentes (para notificar a existência de todos para todos); o IP do líder atual (o remetente da mensagem); e um número inteiro para cada um dos componentes, para que, caso o líder falhe, uma nova rodada para a decisão do próximo líder seja iniciada (o processo cujo número inteiro for a seqüência do número do antigo líder será o próximo líder).

Deste modo, a operação de consenso é iniciada, e os passos especificados no capítulo 2 são realizados, como ilustra a figura 4.8:

- primeira fase: o detector que suspeitou de algum objeto monitorado envia uma mensagem ao líder (caso o próprio não o seja) requisitando o consenso; a seguir, o líder notifica os processos do sistema da necessidade do consenso, com uma mensagem do tipo "*Consensus_Init*"; os processos então enviam suas propostas (sua percepção do estado do suspeito) ao coordenador, em mensagens do tipo "*Propose*";
- segunda fase: o coordenador espera para que a maioria dos processos envie suas propostas;
- terceira fase: os processos esperam pela decisão do coordenador, e quando a recebem (mensagem do tipo "*Decision*"), enviam uma resposta positiva (*ACK*). Caso o coordenador falhe neste período de tempo, o novo coordenador será escolhido. Neste caso, cada detector irá analisar seu número, enviado pelo primeiro líder, e verificar se é o próximo a ser escolhido. Caso sim , notificará o resto dos

processos que é o atual líder, e a operação de consenso é reiniciada;

- quarta fase: o coordenador espera por mensagens da maioria dos processos; se a maioria das mensagens for *ACK*, o coordenador valida a decisão sobre o suspeito e informa aos demais, com mensagens do tipo "*Final_Decision*".

Deste modo, os detectores acabam possuindo uma lista de suspeitos unificada, diminuindo a possibilidade de falsas suspeitas.

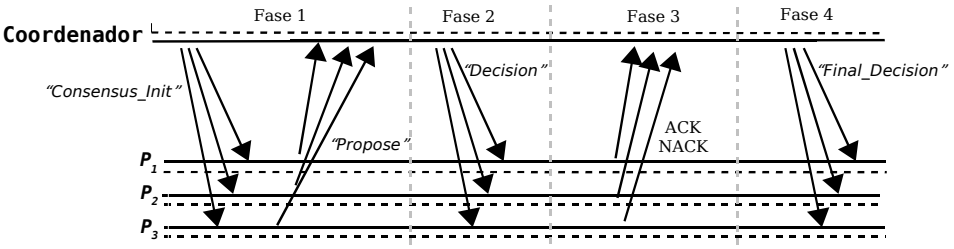

Figura 4.8 – Troca de mensagens no protocolo de consenso

4.7 Empacotamento do Serviço

Originalmente, o AFDService consistia em dois conjuntos de classes (arquivos java com extensão *.class*) contidos em dois diretórios distintos, *FD* e *TS*. Deste modo, para a utilização do serviço era necessário adicionar estes dois diretórios à variável de ambiente *CLASSPATH* do sistema operacional utilizado.

Para tornar a instalação e uso do serviço uma tarefa mais fácil, foi criado o arquivo *AFDService.jar*, que contém todas as classes do serviço de detecção de defeitos. Agora se tornou mais fácil utilizar o serviço, pois é somente necessário adicionar o arquivo *AFDService.jar* no *CLASSPATH*, ao invés dos dois diretórios.

Foi utilizada a IDE NetBeans [NETBEANS, 2004] para gerar os *scripts* ANT [APACHE, 2004a], que possuem tarefas para empacotar o serviço em um arquivo *jar*, além de compilar e compactar todo o serviço.

4.8 Documentação do Serviço

O manual de instalação, documentação, instruções de uso e informações de como fazer o download do AFDService foram descritos em páginas HTML, de modo que usuários possam obter todas as informações sobre o serviço de modo fácil e prático. A figura 4.9 ilustra um *snapshot* da página de apresentação do AFDService.

A página foi dividida em quatro *links*:

- apresentação: introduz o AFDService ao usuário, explanando os propósitos gerais do serviço, para que fins pode ser utilizado e o modelo de sistema utilizado;
- *downloads*: disponibiliza para *download* as versões funcionais do serviço. Esta seção será atualizada assim que outras funcionalidades forem adicionadas ao AFDService;
- instalação: demonstra os passos necessários para instalação do serviço, de forma clara e com exemplos práticos, praticamente sanando dúvidas quanto a instalação;
- documentação: apresenta a documentação do AFDService, como usá-lo, e ilustra o código fonte documentado de exemplos, de forma que o usuário facilmente saiba utilizar as funcionalidades do serviço. Esta seção também será atualizada com o passar do tempo, pois é necessário a adição de novos exemplos mais complexos que ilustrem todo o potencial de uso do ADFService.

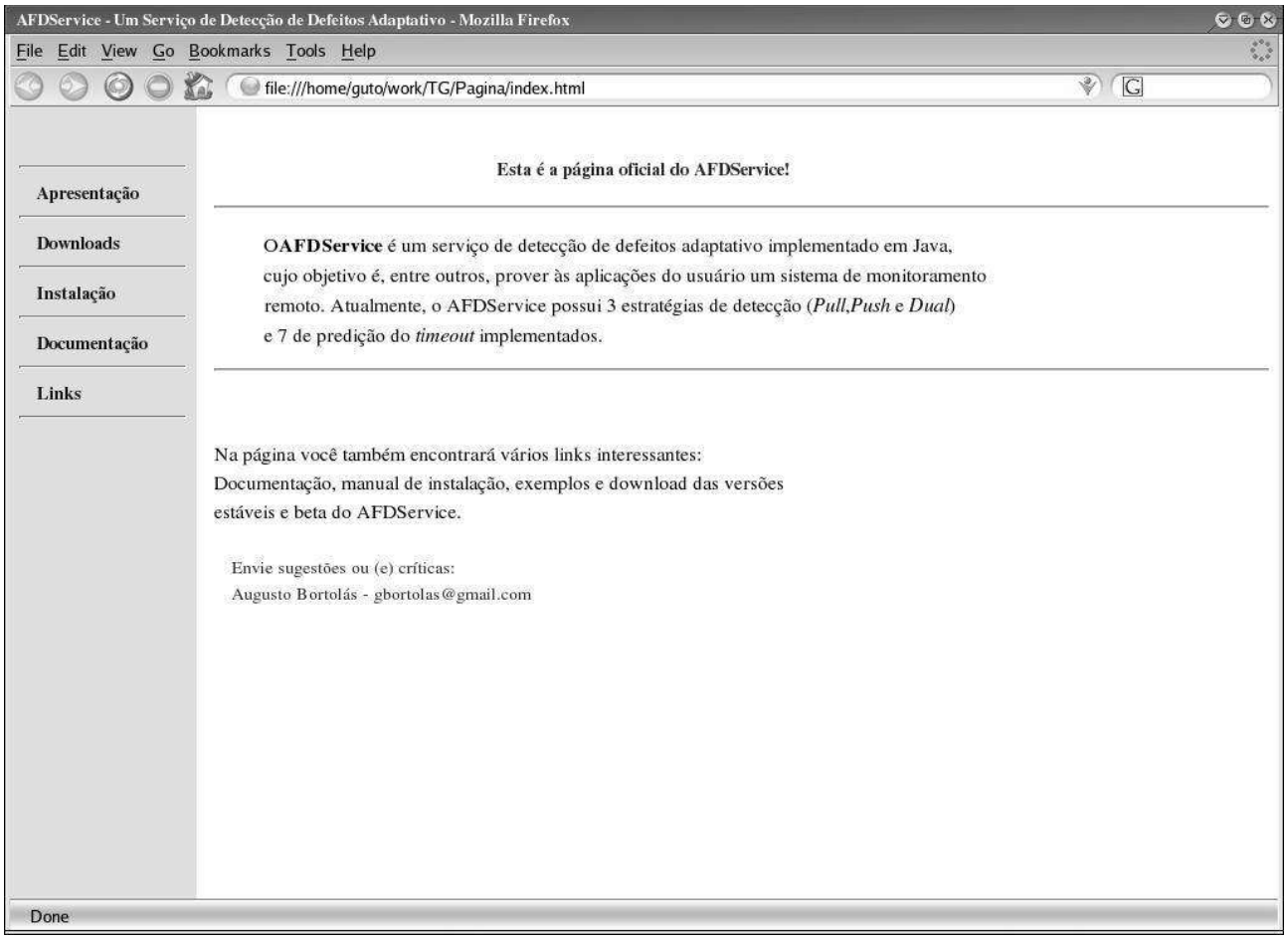

Figura 4.9 – *Snapshot* da página de apresentação do AFDService

5 CONCLUSÕES

Este trabalho apresenta o estudo e implementação de várias funcionalidades do AFDService, tornando-o um serviço mais robusto e completo. Com a agregação destas funcionalidades demonstra-se que: os algoritmos de detecção e o mecanismo para transpor *firewalls* tornam o AFDService um serviço mais adaptável a diferentes ambientes e mais seguro; o protocolo de acordo aumenta a qualidade do serviço, diminuindo as chances de ocorrerem falsas suspeitas por algum dos componentes do sistema; a adoção do padrão de projeto *singleton* torna o serviço mais seguro, diminuindo a probabilidade de ocorrer conflito de instanciação do detector; o empacotamento do serviço torna-o mais fácil e prático de ser instalado e utilizado; a documentação, apresentada em páginas WEB, facilita o aprendizado do AFDService, tanto de uso como de desenvolvimento, e também facilita a busca de informações sobre o mesmo.

Foram realizados testes de desempenho entre detectores usando comunicação UDP e HTTP, e os resultados mostraram que o tempo de comunicação entre detectores usando HTTP é relativamente maior. No entanto, seu uso ainda é válido, pois, além de seu desempenho ser bom o suficiente para sua utilização nos detectores, diminui brechas na segurança das redes, tornando-as mais seguras.

Para o futuro, é interessante realizar mais testes funcionais com a solução para transpor *firewalls*, otimizar os protocolos implementados, de modo que os mesmos diminuam a quantidade de mensagens trocadas na rede, e analisar outras alternativas de comunicação para transpor *firewalls*, como PHP. Também é necessário incrementar a documentação do ADFService, implementando exemplos mais complexos, para que toda a potencialidade do serviço possa ser demonstrada de maneira prática.

6 BIBLIOGRAFIA

[AGGARWAL; GUPTA, 2002] AGGARWAL, Anurag; GUPTA, Diwaker; **Failure Detectors for Distributed Systems**. Indian Institute of Technology, Kanpur, 2002.

[APACHE, 2004a] APACHE Ant. **Apache Ant *Java-based* Build Tool**. Disponível em: <http://ant.apache.org/>. Acesso em: out. 2004.

[APACHE, 2004b] APACHE Tomcat. **Apache Tomcat Servlet Container**. Disponível em: <http://jakarta.apache.org/tomcat/>. Acesso em: out. 2004.

[BORTOLAS; NUNES, 2003] BORTOLÁS, Augusto O.; NUNES, Raul C.; **Transpondo *Firewalls* em Sistemas Distribuídos**. CRICTE, Univali – Itajaí – SC, 2003.

[BORTOLAS; KREUTZ; NUNES, 2004] BORTOLÁS, Augusto O.; KREUTZ, D. L.; NUNES, Raul C. ***Firewalls*: Reflexão Baseada em Exemplos**. Revista do Centro de Ciências da Economia e Informática, CCEI, v.8, n.13, Universidade da Região da Campanha, Março 2004.

[BRASILEIRO; FIGUEIREDO; SAMPAIO, 2002] BRASILEIRO, F. V.; FIGUEIREDO, J. C. A. de; SAMPAIO, L. M. R. **A Hierarquical Failure Detection Service with Perfect Semantics**. Workshop de Testes e Tolerância a Falhas, WTF, 3., 2002, Buzios – Brasil. Anais.. Rio de Janeiro: UFRJ, 2002. p.25-32.

[CHANDRA; TOUEG, 1996] CHANDRA, T. D.; TOUEG, Sam. **Unreliable Failure Detectors for Reliable Distributed Systems**. Journal of the ACM, New York, v.43, n.2, p.225-267, March, 1996.

[CRISTIAN, 1991] CRISTIAN, Flaviu. **Understanding Fault-Tolerant Distributed Systems**. Communications of the ACM, New York, v.43, n.2, p.56-78, Feb., 1991.

[DEITEL, 2001] DEITEL, H. M.; DEITEL; P. J. **Java, Como Programar**, Terceira Edição. Porto Alegre: Bookman, 2001.

[DUANE; MARK, 2000] DUANE, K. Fields; MARK, A. Kolb. **Desenvolvendo na WEB com Java Server Pages**. Rio de Janeiro: Editora Ciência Moderna Ltda., 2000.

[ESTEFANEL; JANSCH-PÔRTO, 2000] ESTEFANEL, Luiz A. B.; JANSCH-PÔRTO, Ingrid; **Detectores de Defeitos Não Confiáveis**. Porto Alegre: Instituto de Informática, janeiro, 2000 (Trabalho Individual de Mestrado).

[ESTEFANEL; JANSCH-PÔRTO, 2001] ESTEFANEL, Luiz A. B.; JANSCH-PÔRTO, Ingrid; **On the Evaluation of Failure Detectors Performance**. IX Brazilian Symposium on Fault-Tolerant Computing (SCTF'2001), Florianópolis – SC – Brasil, 5-7 March 2001.

[FELBER; FAYAD; GUERRAOUI, 1999a] FELBER, Pascal; FAYAD, Mahamed E.; GUERRAOUI, Rachid. **Putting OO Distributed Programming to Work**. Communications of the ACM, New York, v.42, n.11, p.97-101, Nov., 1999.

[FELBER; DÉFAGO; GUERRAOUI, 1999b] FELBER, Pascal; DÉFAGO, Xavier; GUERRAOUI, Rachid; **Failure Detectors as First Class Objects**. IEEE Computer Society Press. Appeared in *Proceedings of the Internacional Symposium on Distributed Objects and Applications (DOA'99)*, 1999.

[FISCHER; LYNCH; PATERSON, 1985] FISCHER, M.; LYNCH, N.; PATERSON, M. **Impossibility of Distributed Consensus With one Faulty Process**. Journal of the ACM, New York, v.32, p.374-382, Apr. 1985.

[GAMMA; HELM; JOHNSON; VLISSIDES, 1994] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns, Elements of Reusable Object-Oriented Software**. Massachusetts: Addison Wesley, 1994. 395 p.

[GARTNER, 1999] GÄRTNER, Felix C.; **Fundamentals of Fault-Tolerant Distributed Computing in Assinchronous Environments**. ACM Computing Surveys, Vol. 31, No. 1, March 1999.

[GUERRAOUI; SCHIPER, 1997] GUERRAOUI, R.; SCHIPER, A. **A Software-Based Replication For Fault Tolerance**. IEEE Computer, Los Alamitos, v.30, n.4 p.68-74, Apr. 1997.

[JALOTE 1994] JALOTE, P. **Fault-Tolerance in Distributed Systems**. New Jersey: Prentice Hall, 1994.

[LARREA; ARÉVALO; FERNÁNDEZ, 1999] LARREA, M.; ARÉVALO, S.; FERNÁNDEZ, A. **Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems**. International Symposium on Distributed Computing, DISC, 13., 1999, Bratislava, Slovak Rep. Proceedings... Berlin: Springer-Verlag, 1999. p.34-48. (Lecture Notes in Computer Science,v.1693).

[LINS, 2002] LINS, Júlio C.; **Automação com Ant, Builds Integração Contínua e Ant 2**. Revista Java Magazine, Edição 2, ano 1, Curitiba – PR, 2002.

[LOZANO, 2003] LOZANO, Fernando; **Tira-Dúvidas – Pacotes JAR e WAR**. Revista Java Magazine, edição 9, ano II, Curitiba - PR, 2003.

[NETBEANS, 2004] NETBEANS. **NetBeans Open Source IDE**. Disponível em: <http://www.netbeans.org/>. Acesso em: out. 2004.

[NUNES; JANSCH-PÔRTO, 2003] NUNES, R. C; JANSCH-PÔRTO, I. **Adaptação Dinâmica do Timeout de Detectores de Defeitos Através do uso de Séries Temporais**. Porto Alegre: Instituto de Informatica - UFRGS, setembro, 2003 (Tese de doutorado).

[PETER, 2004] PETER, J. J. **Padrões de Projeto em Java, Reutilizando o Projeto de Software**. Revista Mundo Java, n. 6, ano I, Curitiba - PR, 2004.

[SERGENT; DÉFAGO; SCHIPER, 1999] SERGENT, N.; DÉFAGO, X; SCHIPER, A. **Failure Detectors: Implementation Issues and Impact on Consensus Performance**. Lausanne: École Polytechnique Fédérale de Lausanne – EPFL, 1999. (Technical Report SSC/1999/019).

[SUN, 2004] SUN Microsystems. **The Source for Java Technology**. Disponível em: <http:\\www.java.sun.com>. Acesso em: out. 2004.

[TANENBAUM; STEEN, 2002] TANENBAUM, Andrew S.; STEEN, Maarten Van. **Distributed Systems: Principles and Paradigms**. New Jersey: Prentice Hall, 2002. 803p.

[TANENBAUM, 2003] TANENBAUM, Andrew S. **Redes de Computadores**. Rio de Janeiro: Editora Campus, 2003.

[URBAN; DÉFAGO; SCHIPER, 2002] URBÁN, P.; DÉFAGO, X.; SCHIPER, A.; **Neko: A Single Environment to Simulate and Prototype Distributed Algorithms**. Journal of Information Science and Engineering 18, 981-997, 2002.

[URBAN; HAYASHIBARA; SCHIPER; KATAYAMA, 2004] URBÁN, P; HAYASHIBARA, N; SCHIPER, A; KATAYAMA, T; **Performance Comparison of a Routing Coordinator and a Leader Based Consensus Algorithm**. 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04), Florianpolis – Brazil, 2004;