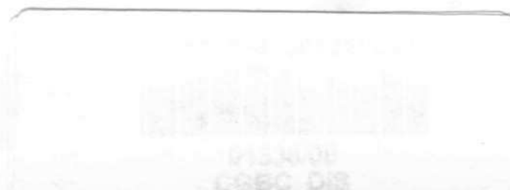
Universidade Federal da Paraíba – UFPB
Centro de Ciências e Tecnologia - CCT
Departamento de Sistemas e Computação - DSC
Coordenação de Pós-Graduação em Informática - COPIN

Tolerância a Falhas em Java através de Comunicação em Grupo

por

LILIANNE DANTAS CIRNE

Campina Grande, Dezembro de 1999



LILIANNE DANTAS CIRNE

Tolerância a Falhas em Java através de Comunicação em Grupo

Dissertação de Mestrado submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba – Campus II como requisito parcial para a otenção do grau de Mestre em Informática.

Orientador: **Raimundo José de Araújo Macêdo**

Área de Concentração: **Ciência da Computação**

Campina Grande, Dezembro de 1999

C578t Cirne, Lilianne Dantas
Tolerancia a falhas em java atraves de comunicacao em grupo / Lilianne Dantas Cirne. - Campina Grande, 1999.
120 f.

Dissertaca (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Sistemas Distribuidos 2. Tolerancia a Falhas 3. Linguagem Java 4. Dissertacao - Informatica I. Macedo, Raimundo Jose de Araujo II. Universidade Federal da Paraiba - Campina Grande (PB)

CDU 004.75(043)

TOLERÂNCIA A FALHAS EM JAVA ATRAVÉS DE COMUNICAÇÃO EM GRUPO

LILIANNE DANTAS CIRNE

DISSERTAÇÃO APROVADA EM 13.12.1999

PROF. RAIMUNDO JOSÉ DE ARAÚJO MACÊDO, Ph.D
Orientador

PROF. PAULO ROBERTO FREIRE CUNHA, Ph.D
Examinador

PROF. JACQUES PHILIPPE SAUVÉ, Ph.D
Examinador

CAMPINA GRANDE – PB.

Agradecimentos

Agradeço aos meus familiares pelo apoio constante; aos meus pais Lúcia e Lindenberg pelo suporte incondicional e determinante para a realização deste trabalho, meus irmãos Lucianne e Lincoln e minha vó Barbara por tornar minha temporada em Campina Grande maravilhosa.

Agradeço ao Professor Raimundo Macêdo pela orientação no desenvolvimento deste trabalho e pelo acompanhamento e incentivo na sua realização. Aos membros e ex-membros do LaSiD que fazem deste laboratório um ótimo ambiente de trabalho: Socorro, Deise, Criston, George, João Bulhões, sempre me ajudando nas dúvidas e Luiza, companheira em todos os momentos.

Às minhas amigas Sueli e Paula que tornaram os momentos de lazer ainda mais agradáveis.

Agradeço à CAPES pela concessão da bolsa de mestrado.

Aos funcionários e professores da Coordenação de Pós-Graduação em Informática da UFPB.

Resumo

O intenso uso dos Sistemas Distribuídos em aplicações de diversas naturezas tem levado a uma necessidade cada vez maior de aplicações confiáveis. Esses sistemas devem oferecer suporte a Tolerância a Falhas, ou seja, os sistemas não devem interromper seu funcionamento mesmo na presença de falhas de alguns componentes (*hardware* ou *software*) e dados. A técnica de replicação ativa de componentes é freqüentemente utilizada quando se quer implementar tais serviços de alta disponibilidade e tolerantes a falhas. Alguns Sistemas de Comunicação em Grupo já oferecem um suporte para a construção de aplicações distribuídas tolerantes a falhas; entretanto, a maioria desses sistemas não são portáveis, uma característica muito importante em sistemas distribuídos.

Neste contexto, a linguagem Java vem adquirindo grande importância nos últimos anos, devido principalmente à sua característica de portabilidade e suporte ao desenvolvimento de aplicações distribuídas. No entanto, a linguagem Java não oferece suporte a Tolerância a Falhas que permita a um serviço distribuído continuar funcionando corretamente caso alguns de seus componentes falhem.

Este trabalho apresenta uma proposta para adição de Tolerância a Falhas ao ambiente Java capaz de suprir as necessidades da replicação ativa. O sistema desenvolvido, denominado iBusTF (iBus Tolerante a Falhas), acrescentou ao iBus novas propriedades de Comunicação em Grupo necessárias para manter a consistência num grupo de replicas ativas: *ordenação total* e *membership atômico*. A abordagem adotada tem a vantagem de somente usar recursos já disponíveis em JAVA, mantendo total compatibilidade com o sistema iBus.

Abstract

The intense use of Distributed Systems in different kinds of applications demands more reliability of these applications. The systems should be prepared to tolerate faults, i.e., a system should not interrupt its operation in the presence of some components (hardware or software) and data faults. Active replication is usually used when one aims at building such high available and fault-tolerant services. Some Group Communication Systems already offer support for the development of fault tolerant distributed applications. However, most of those systems are not portable, a very important property in distributed systems.

In this context, the Java language has become widely used in Distributed Systems in the last years, specially due to its portability and facilities for the development of distributed applications. Nonetheless, Java provides no support for the development of fault-tolerant distributed applications which can continue to function properly despite component failures.

This paper describes an approach for fault-tolerance in Java which can meet the requirements of active replication. In order to achieve that, an extension to the iBus package designed by Silvano Maffeis [MAF96] has been developed and implemented. The developed system, named iBusTF (fault-tolerant iBus), added new group communication properties required by active replication : *total order delivery and atomic membership*. The approach adopted has the advantage of only using Java resources, keeping total compatibility with the iBus system.

Sumário

Lista de Abreviaturas IV

Lista de Figuras V

Lista de Tabelas VII

Capítulo 1 – Introdução 1

1.1. Estrutura da Dissertação 07

Capítulo 2 – Comunicação em Grupo em Sistemas Distribuídos 8

2.1. Introdução 9

2.2. Propriedades Fundamentais da Comunicação em Grupo 10

2.2.1. Endereço de Grupo 10

2.2.2. Atomicidade 10

2.2.3. Ordenação 11

2.2.4. Reconfiguração (*Membership*) 13

2.2.4.1. Sincronismo Virtual 13

2.3. Sistemas e Plataformas de Grupos 15

2.3.1. ISIS 15

2.3.2. Horus 17

2.3.3. Base Confiável de Comunicação em Grupo(BCG) 20

2.3.4. Comunicação em Grupo em CORBA 22

2.3.4.1. Breve Introdução a CORBA 22

2.3.4.2. Comunicação em Grupo na Plataforma CORBA 25

2.4. Conclusão 27

Capítulo 3 – Adicionando Tolerância a Falhas a Java através de Comunicação em Grupo 29

3.1. Introdução 30

3.2. Aspectos da Comunicação entre Objetos na Linguagem Java 32

3.2.1.Protocolos de Comunicação TCP e UDP 33
3.2.2.RMI (*Remote Method Invocation*) 34
3.3.Tolerância a Falhas em Java através de Comunicação em Grupo 37
3.3.1. Estendendo a Classe *MulticastSocket* 40
3.3.2.Estendendo o RMI 42
3.3.3.Utlizando o Serviço de uma Plataforma CORBA com Suporte a Grupos 43
3.3.4.Solução Adotada para Tolerância a Falhas em Java 44
3.4.iBus (*Java Intranet Software Bus*) 46
3.4.1. Modelo de Comunicação 48
3.4.2.Camadas de Protocolos do iBus 51
3.5.Conclusão 57
Capítulo 4 – Adicionando Tolerância a Falhas ao iBus 60
4.1.Introdução 61
4.2.iBusTF (iBus Tolerante a Falhas) 62
4.2.1.Protocolo de Ordenação Total 63
4.2.2.Protocolo de *membership* atômico 66
4.3.Detalhes da Implementação 71
4.3.1.Classe TFclass 75
4.3.1.1.Operação *dnInit* 77
4.3.1.2.Operação *dnPush* 77
4.3.1.3.Operação *upHandleEvent* 77
4.3.1.4.Operação *Agreement* 78
4.3.1.5.Operação *updateView* 79
4.3.2.Classe *BlockMatrix* 79
4.3.3.Classe *TimeOut* 81
4.3.4.*Thread* LTS 81
4.3.5.*Thread* Delivery 82
4.3.6.Classe *HandleMembers* 84

4.3.7. Classe *MessageNull* ... 85

4.3.8. Classe *Suspect* ... 85

4.3.9. Classe *Refute* ... 86

4.3.10. Classe *Recovery* ... 87

4.3.11. Classe *Confirmed* ... 88

4.3.12. Classe *Semaphore* ... 88

4.4. Dados de Desempenho ... 89

4.5. Conclusão ... 94

Capítulo 5 – Comparação do iBusTF com outras propostas para Tolerância a Falhas em Java ... 96

5.1. Introdução ... 97

5.2. Filterfresh ... 97

5.3. ROAPI ... 103

5.4. JavaGroups ... 104

5.5. Conclusão ... 107

Capítulo 6 – Conclusão ... 110

6.1. Trabalhos Futuros ... 115

Referências Bibliográficas ... 116

Lista de Abreviaturas

BCG	Base Confiável de Comunicação em Grupo
BM	Block Matrix
bn	Block Number
CORBA	Commom Object Request Broker Architecture
FIFO	First In First Out
FTP	File Transfer Protocol
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
iBus	Java Intranet Software Bus
IP	Internet Protocol
IDL	Interface Definition Language
LaSiD	Laboratório de Sistemas Distribuídos
lcb	Last Complete Block
LRV	Last Received Vector
Newtop	Newcastle Total Order Protocol
OMG	Object Management Group
ORB	Object Request Broker
RMI	Remote Method Invocation
ROAPI	Replicated Object API
RPC	Remote Procedure Call
SV	Stability Vector
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UFBA	Universidade Federal da Bahia
UFPB	Universidade Federal da Paraíba
WWW	World Wide Web

Lista de Figuras

FIGURA 1.1: Modelo dos Sistemas Distribuídos....2
FIGURA 1.2: Hierarquia de Falhas....4
FIGURA 2.1 Ordenação Causal de mensagens....12
FIGURA 2.2: Ordenação Total de mensagens....12
FIGURA 2.3: Mensagens são entregues na mesma visão em que foram enviadas....13
FIGURA 2.4: As mensagens não são entregues na visão em que foram enviadas....14
FIGURA 2.5: Primitiva CBCAST do ISIS....17
FIGURA 2.6: Arquitetura de Camadas do Horus....18
FIGURA 2.7: Camadas da plataforma BCG....20
FIGURA 2.8: Arquitetura da BCG....22
FIGURA 2.9: Arquitetura da plataforma CORBA....24
FIGURA 2.10: Modelo Orbix + ISIS....26
FIGURA 2.11: Serviço de Comunicação em grupo em CORBA....26
FIGURA 3.1: Comunicação através da classe *MulticastSocket*....33
FIGURA 3.2: Serviço de Nomes do RMI....34
FIGURA 3.3: Arquitetura do RMI....35
FIGURA 3.4: Tolerância a Falhas em Java utilizando a classe *MulticastSocket*....41
FIGURA 3.5: Replicação de objetos servidores no RMI....42
FIGURA 3.6: Objetos Java utilizando um Serviço de Comunicação em Grupo da plataforma CORBA....44
FIGURA 3.7: Arquitetura do iBus....47
FIGURA 3.8: Modelo de comunicação *Publish/Subscribe*....48
FIGURA 3.9: Parte do código do Transmissor....49
FIGURA 3.10: Parte do código do Receptor....50
FIGURA 3.11: Operações da classe abstrata *ProtocolObject*....51
FIGURA 3.12: Mudanças de visão geradas pela camada REACH....53
FIGURA 3.13: Inconsistência nas visões das aplicações....54

FIGURA 4.1: Uma Matriz de Blocos com seis objetos, membros de um mesmo grupo....64
FIGURA 4.2: Arquitetura da camada TF....66
FIGURA 4.3: Suspeita incorreta de falha de um membro....68
FIGURA 4.4: Acordo sobre a suspeita de falha de um membro....70
FIGURA 4.5: Principais classes da camada TF....71
FIGURA 4.6: Diagrama de seqûencia da camada TF....72
FIGURA 4.7: Operações e atributos da classe *Event*....74
FIGURA 4.8: Classes que compõem a camada TF....75
FIGURA 4.9: Atributos e operações da classe TF....76
FIGURA 4.10: Operações da classe *BlockMatrix*....78
FIGURA 4.11: Operação *dnRegisterTalker*....79
FIGURA 4.12 *Threads* do iBusTF....80
FIGURA 4.13: Operações da classe *BlockMatrix*....80
FIGURA 4.14: Operações da classe *TimeOut*....81
FIGURA 4.15: Operações da classe LTS....82
FIGURA 4.16: *Threads* do iBusTF....83
FIGURA 4.17: Operações da classe *MessageNull*....85
FIGURA 4.18: Operações da classe *Suspect*....86
FIGURA 4.19: Operações da classe *Refute*....87
FIGURA 4.20: Operações da classe *Recovery*....87
FIGURA 4.21: Operações da classe *Confirmed*....88
FIGURA 4.22: Operações da classe *Semaphore*....89
FIGURA 4.23: Diagrama de sequências do *clientTimeRT*....91
FIGURA 4.24: Escalabilidade segundo *Round Trip* no iBusTF....92
FIGURA 4.25: Escalabilidade segundo *Round Trip* no iBus....92
FIGURA 4.26: Gráfico do tempo médio para realização do consenso....94
FIGURA 5.1: Operação de atualização no *FT Registry*....100
FIGURA 5.2: Replicação de objetos servidores no *Filterfresh*....101

FIGURA 5.3: Mensagem de erro de um objeto servidor interceptada na camada RRL....102
FIGURA 5.4: Chamada a um objeto servidor replicado....102
FIGURA 5.5: Estrutura de um grupo no ROAPI....103
FIGURA 5.6: Arquitetura do JavaGroups....105
FIGURA 5.7: Estados de um canal....106

Lista de Tabelas

TABELA 3.1: Comparativo entre as abordagens........39

TABELA 3.2: Algumas características das camadas do iBus........56

Capítulo 1

Introdução

Neste capítulo será apresentada, a partir de uma breve descrição de Sistemas Distribuídos, Tolerância a Falhas e da linguagem Java, a motivação para o presente trabalho. O capítulo é finalizado com a estrutura da Dissertação.

Os Sistemas Distribuídos são geralmente caracterizados por um conjunto de objetos (processos ou *software* em execução), localizados em estações de trabalho independentes (PCs, Servidores, *Laptops*, etc.), onde os objetos se comunicam por troca de mensagens através de uma rede de comunicação (figura 1.1). O desenvolvimento dos Sistemas Distribuídos foi impulsionado principalmente pelo surgimento das redes de computadores nos anos 70, pelo barateamento dos computadores pessoais, tornando-se de fácil aquisição para uma grande fatia da população mundial e pelo desenvolvimento de *software* para sistemas distribuídos.

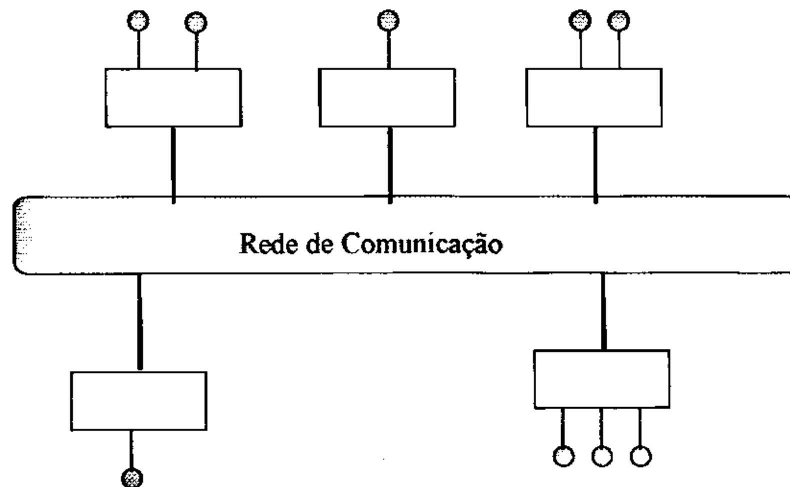


Figura 1.1: Modelo dos Sistemas Distribuídos

Os Sistemas Distribuídos apresentam algumas vantagens sobre sistemas centralizados tais como: *compartilhamento de recursos*; *paralelismo*, a presença de diversas estações de trabalho possibilita a execução de tarefas em paralelo, contribuindo para um aumento de desempenho, e a *flexibilidade* para adicionar novos componentes, sem interrupções no funcionamento do sistema[CDK94]. No entanto, existem alguns problemas relacionados aos Sistemas Distribuídos como a complexidade no desenvolvimento de software para sistemas distribuídos, saturação na rede de comunicação e principalmente falhas dos componentes. Se o sistema distribuído não estiver preparado para "tolerar falhas", a falha de um componente pode comprometer o funcionamento do sistema como um todo.

O objetivo da tolerância a falhas é garantir a continuidade do serviço oferecido mesmo na presença de falhas. Por ser um ponto importante no contexto dessa dissertação, apresentaremos os conceitos de erro, falha e defeito. Utilizaremos a terminologia *fault* para designar falha, *error* para erro e *failure* para designar defeito.

Esses termos têm significados diferentes do ponto de vista de tolerância a falhas. *Falha* pode ser causada por software (problemas de especificação e implementação), hardware (componentes defeituosos) ou agentes externos (radiação, interferência eletromagnética, entre outros)[JAL94]. O *erro* é causado por uma falha. Se existe um erro no sistema, então existe uma seqüência de ações que podem ser executadas pelo sistema e que levarão a um *defeito*, a não ser que técnicas de Tolerância a Falhas sejam empregadas [JAL94]. Um componente é considerado defeituoso quando não atende às especificações para o qual foi projetado.

As falhas são classificadas levando-se em consideração o sistema de comunicação adotado, síncrono ou assíncrono. Nos sistemas síncronos há um limite de tempo conhecido para a execução de ações. Nos sistemas assíncronos não há um tempo máximo para que as ações sejam executadas, e o tempo de transmissão das mensagens não é conhecido. Nesses sistemas é impossível distinguir entre a falha de um membro e atrasos nos canais de comunicação.

Nos sistemas síncronos as falhas podem ser divididas em quatro categorias: *fail-stop*, omissão, performance e bizantinas[JAL94]. Uma falha é do tipo *fail-stop* quando um componente pára de funcionar e não emite mais nenhuma resposta. Uma falha de *omissão* ocorre quando um componente não responde às solicitações recebidas, e uma falha de *performance* ocorre quando um componente responde às solicitações fora do intervalo de tempo especificado. Uma falha *bizantina* ou arbitrária ocorre quando um componente responde a uma solicitação de maneira diferente ao qual foi especificado. As quatro categorias de falhas formam um hierarquia, como mostra a figura 1.2. A falha do tipo *fail-stop* é a mais restritiva e a falha bizantina engloba os outros tipos de falhas.

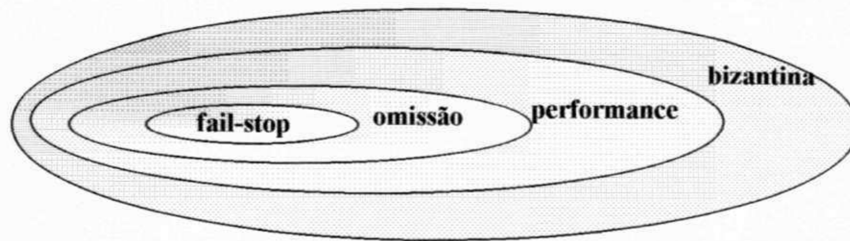


Figura 1.2: Hierarquia de falhas nos sistemas síncronos

Nos sistemas assíncronos as falhas são divididas em *crash*, omissão e bizantina. Em relação à hierarquia da figura 1.2, nos sistemas assíncronos não há falhas de performance e falhas do tipo *fail-stop* são denominadas *crash*. Nessa dissertação, adotamos o modelo assíncrono e falhas do tipo *crash*, utilizado na plataforma BCG[MACa95, MACb95, MAC98, GM98, LMa99] e no sistema iBus[MAF96].

As técnicas utilizadas para prover Tolerância a Falhas caracterizam-se pela redundância, ou seja, a presença de componentes (*Hardware e Software*) ou informações redundantes. Cada membro do grupo constitui-se de um componente do sistema, e deve ser individualmente confiável, garantindo que o serviço continuará a ser oferecido dentro das suas especificações mesmo na presença de falhas de alguns componentes. Em função da técnica de replicação utilizada e para garantir a consistência do grupo de réplicas, a atualização de um componente deve ser seguida pela atualização de todas as réplicas, através de propriedades da Comunicação em Grupo, como difusão confiável (uma mensagem enviada é recebida por todos os membros) e ordenação de mensagens.

Em especial, a replicação ativa de componentes *(active replication)*[SCH90] é freqüentemente utilizada quando se deseja implementar serviços distribuídos de alta disponibilidade e tolerantes a falhas. Nessa técnica de replicação, as mensagens enviadas devem ser percebidas pelas réplicas na mesma ordem global (*ordenação total*) e as mudanças na composição do grupo (entradas, saídas e falhas) devem ser percebidas numa ordem mutuamente consistente entre as réplicas (*membership atômico* – ver capítulo 2).

membros não suspeitos de falha, já suspeitaram da falha de O_k. O_i envia então um evento de confirmação (*Confirmed*), indicando que o acordo foi alcançado. Os membros invocam então a operação *updateView()* para remover O_k do grupo.

4.3 – Detalhes da Implementação

A camada TF implementada neste trabalho é composta por um conjunto de classes. Nessa camada encontram-se implementadas as operações comuns a todas as camadas do iBus e as operações para a realização do acordo sobre a falha de membros e para mudança da visão do grupo. A camada TF também manipula sete tipos de eventos: *Message*, *View*, *MessageNull*, *Suspect*, *Refute*, *Recovery* e *Confirmed*, todos herança da classe *iBus.Event* (figura 4.5).

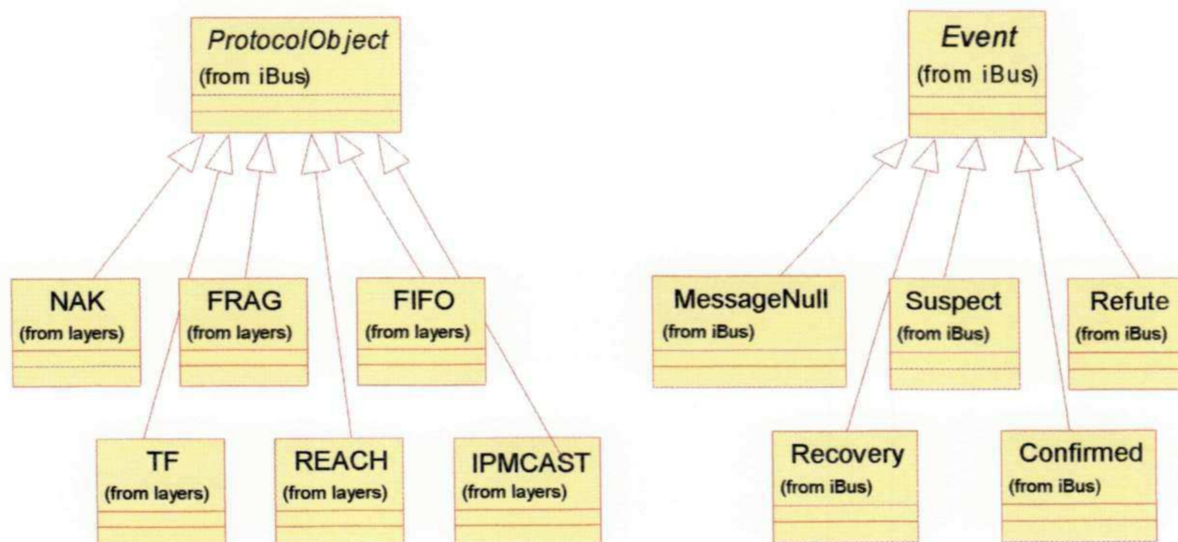


Figura 4.5: Principais classes do iBusTF

A figura 4.5 apresenta, seguindo o padrão UML, as camadas que compõem o iBusTF e os eventos gerados nesta camada. Todas as camadas que compõem o iBusTF são herança da classe *ProtocolObject*, e os eventos gerados na camada TF são herança da classe Event. Os eventos gerados na camada TF são o *evMessageNull*, *evSuspect*, *evRefute*, *evRecovery* e *evConfirmed*. Os outros eventos são gerados pelas outras camadas do iBusTF.

A classe principal da camada TF também é denominada TF. Nessa classe encontram-se implementadas as operações comuns a todas as camadas: *dnPush*, *upHandleEvent*, *dnInit*, *dnSubscribe*, *dnUnsubscribe*, *dnRegisterTalker* e *dnUnregisterTalker*. Essas operações são a interface entre a camada TF e as outras camadas do iBusTF (figura 4.6). Na classe TF também econtram-se implementadas as operações de *Agreement* e *updateView*, para realização do consenso sobre a falha de membros e atualização da visão do grupo, respectivamente

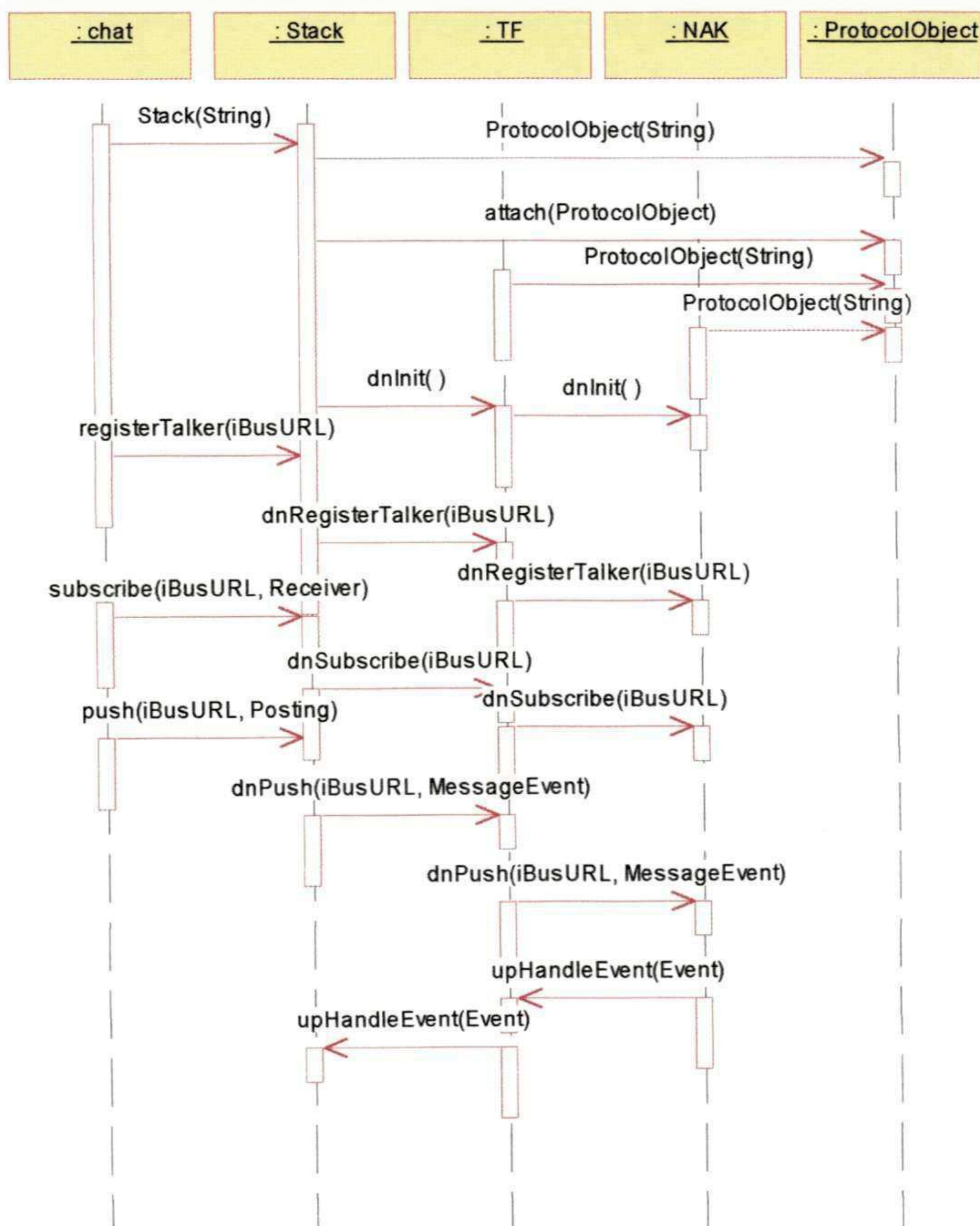


Figura 4.6: Diagrama de sequência da camada TF

A figura 4.6 mostra o diagrama de sequência da camada TF. Uma aplicação do tipo *chat* (envia e recebe mensagens a um grupo de objetos) se comunica com a classe Stack, que é a interface entre a aplicação e as outras camadas do iBusTF. Inicialmente são especificadas as camadas que irá compor a pilha, passando uma String como parâmetro. Se a aplicação desejar utilizar por exemplo todas as camadas do iBusTF, então a string será Stack ("TF(GroupSize=4):FRAG:NAK:FIFO:REACH:IPMCAST"), onde *GroupSize* é o tamanho do grupo a ser criado. A classe *Stack* conecta as camadas que irão compor a pilha (todas herança da classe *ProtocolObject*) e inicializa então estas camadas (operação *dnInit*). Para que a aplicação *chat* possa enviar mensagens ao grupo, este deve se registar no grupo (*registerTalker*) e para receber mensagens do grupo, deve chamar a operação *subscribe*. Estas operações são propagadas às outras camadas. Após estes procedimentos, a aplicação *chat* pode então enviar mensagens ao grupo através da operação *push*. As mensagens trafegam no sentido contrário da pilha através de chamadas a operação *upHandleEvent*.

A classe *Event* define todos os eventos que serão manipulados no iBus. Ao adicionarmos novos eventos do iBus, gerados pela camada TF, estes eventos também foram informados na classe *Event*. Estes eventos são identificados por um valor inteiro(figura 4.7).

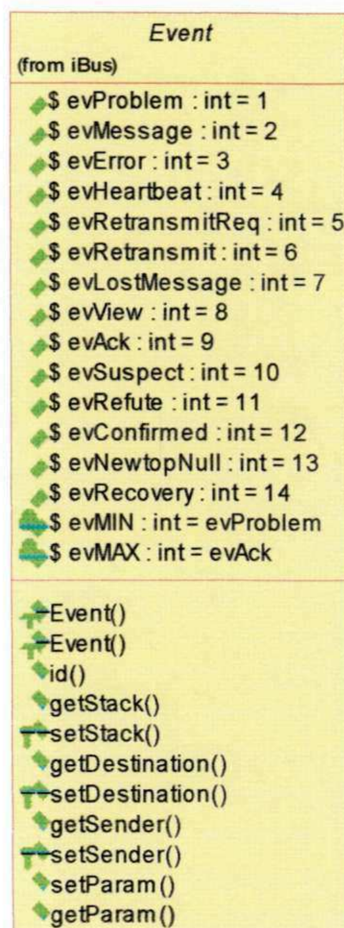


Figura 4.7: Operações e Atributos da classe *Event*

A camada TF é formada por 7 classes principais. A classe *TimeOut* é a classe básica usada pelo LTS (*Local Time Silence*). A *thread* LTS é responsável pelo envio de mensagens nulas com o objetivo de completar blocos na Matriz de Blocos. A classe *HandleMembers* atribui identificadores únicos aos membros dos grupos. Na classe *BlockMatrix* encontram-se as operações para manipulação da Matriz de Blocos (criação da matriz, inserção de mensagens, etc.). A *thread Delivery* é responsável pela remoção das mensagens da Matriz de Blocos e envio destas para a aplicação (figura 4.8).

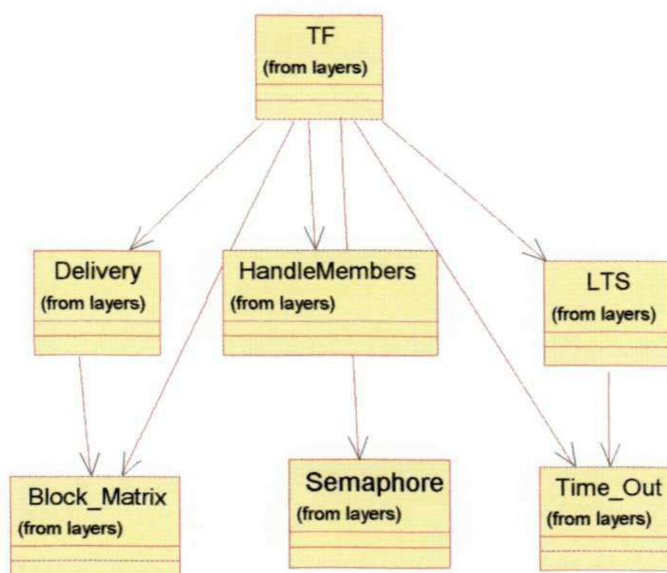


Figura 4.8: Classes que compõem a camada TF

As classes que compõem a camada TF são descritas em mais detalhes a seguir.

4.3.1 – Classe TF

A classe TF é a principal da camada TF, onde encontram-se as operações presentes em todas as camadas, isto é, para introduzir uma nova camada na arquitetura do iBus, é necessário implementar as operações comuns à todas as camadas do iBus que são: *dnInit*, *dnPush*, *dnSubscribe*, *dnRegisterTalker*, *dnUnsubscribe*, *dnUnRegisterTalker* e *UpHandleEvent*. Essas operações são necessárias para que uma camada possa ser empilhada sobre a outra e os eventos sejam enviados entre elas. As outras operações implementadas na classe TF são *Agreement* e *updateView* (figura 4.9).

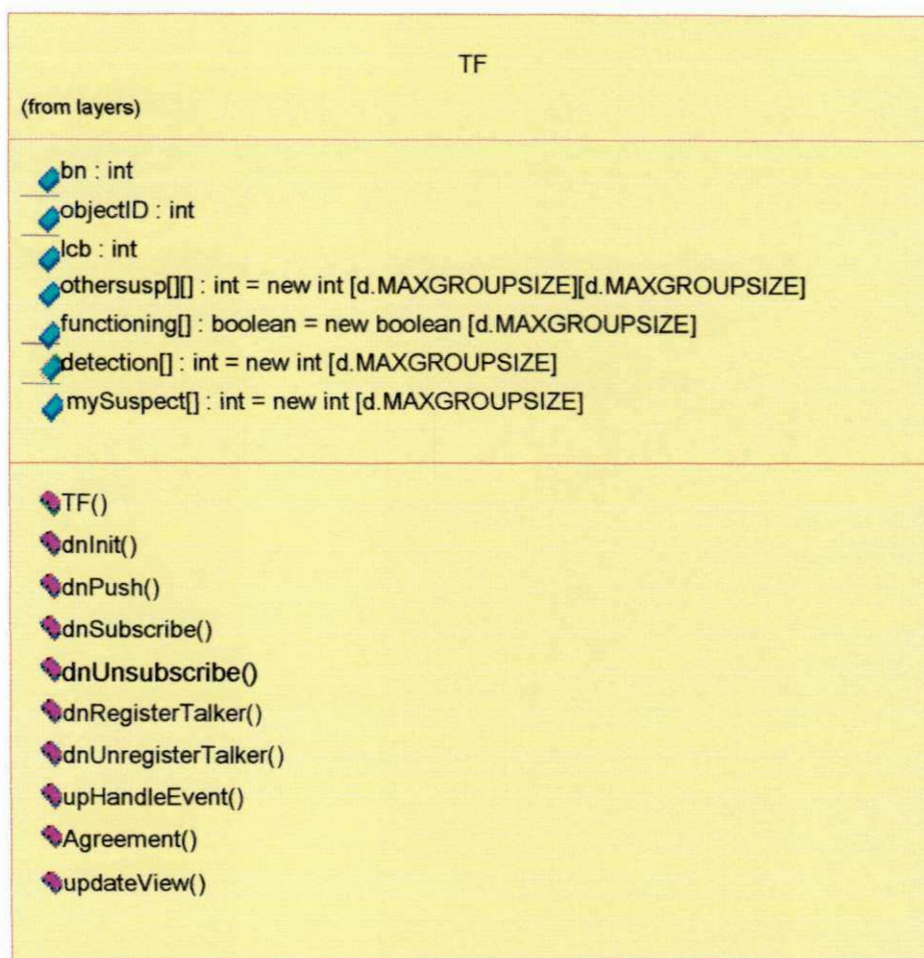


Figura 4.9: Atributos e Operações da classe TF

A figura 4.9 apresenta os atributos e as operações da classe TF. As variáveis *bn*, *objectID* e *lcb* armazenam as informações que serão enviadas no cabeçalho das mensagens. *bn* é o numero de bloco da mensagem, *objectID* é o identificador de cada objeto membro do grupo e *lcb* contém o valor do último bloco completo da Matrix de Blocos. O vetor *functioning* armazena o *objectID* dos objetos operacionais do grupo. O vetor *mySuspect* armazena o *bn* das mensagens dos objetos suspeitos de falha e o vetor *otherSuspect* armazena o *bn* das mensagens dos objetos suspeitos de falha pelos outros membros do grupo. Os vetores possuem o mesmo tamanho do grupo (MAXGROUPSIZE). As operações da classe TF são descritas em mais detalhes a seguir.

4.3.1.1 – Operação *dnInit*

A operação *dnInit* inicializa a camada TF juntamente com as estruturas de dados e *threads*. Esta operação é executada depois que a pilha é formada e as camadas são conectadas.

4.3.1.2 – Operação *dnPush*

A operação *dnPush* é usada para enviar os eventos de cima para baixo da pilha de camadas. A mensagem, após ser enviada pela aplicação, é recebida pela camada STACK sendo enviada desta para a camada TF através da operação *dnPush*. Na camada TF a mensagem é desempacotada e são adicionadas novas informações ao cabeçalho: um número de bloco associado a cada mensagem (*bn*), o número do último bloco completo na Matriz (*lcb*) e o identificador do objeto que está enviando a mensagem (*objectID*). Esse identificador único é necessário para localizar o objeto nas estruturas de dados (Matriz e vetores) de cada um dos outros membros do grupo. A mensagem é novamente empacotada e enviada para a camada inferior através da operação *dnPush*.

4.3.1.3 – Operação *upHandleEvent*

Os eventos quando chegam na camada IPMCAST, são enviados para a camada superior através da operação *upHandleEvent*. Esta operação é responsável por transmitir os eventos no sentido de baixo para cima da pilha de camadas. Diferentes tipos de eventos são tratados na camada TF, todos herança da classe iBus.*Event*. Os eventos enviados pelas camadas são, em sua maioria, utilizados para troca de informações entre as camadas correspondentes, e não são entregues à aplicação (com exceção do *ev.Message* e do *ev.View*). Os eventos utilizados na camada TF são mostrados abaixo.

```
public void upHandleEvent(Event event)
{
    switch(event.id( )) {
    //Mensagem enviada pela aplicação
    case Event.evMessage:
    MessageEvent msg = (MessageEvent)event;
    ...
    //Mensagem nula enviada pelo LTS
    case Event.ev.MessageNull:
    MessageNull nul = (MessageNull)event;
    ...
    // Evento Visão enviado pela camada REACH
    case Event.evView:
    View v = (View)event;
    ...
    // Evento Suspeita enviado pela camada TF
    case Event.evSuspect:
    Suspect susp = (Suspect)event;
    ...
    // Evento Refute enviado pela camada TF
    case Event.evRefute:
    Refute ref = (refute)event;
    ...
    // Evento Confirmação enviado pela camada TF
    case Event.evConfirmed:
    Confirmed conf = (Confirmed)event;
    ...
    // Evento Recuperação enviado pela camada TF
    case Event.evRecovery:
    Recovery rec = (Recovery)event;
    ...
    }
}
```

Figura 4.10: Os eventos tratados na camada TF

4.3.1.4 – Operação *Agreement*

Esta operação realiza o acordo sobre a suspeita de falha dos membros, baseado no evento Visão (*evView*) com a suspeita de falha de um membro, enviado pela camada REACH. O acordo sobre a suspeita de falha é realizado comparando o vetor *mySuspect* com o vetor *otherSuspect*. Se para cada entrada de *mySuspect*, também houver uma entrada correspondente em *otherSuspect* para todos os membros operacionais, isto é, pertencentes ao vetor *functioning*, o(s) membro(s) é(são) considerado(s) falho(s) e posteriormente removido(s) do grupo.

4.3.1.5 – Operação *updateView*

Esta operação recebe como parâmetro o vetor *detection* contendo os membros considerados falhos após o acordo, e atualiza a visão do grupo removendo o(s) membro(s) das estruturas de dados *mySuspect*, *otherSuspect*, *BlockMatrix*, *functioning*, LRV e SV.

As outras operações obrigatórias em todas as camadas, *dnSubscribe*, *dnRegisterTalker*, *dnUnsubscribe* e dn*UnregisterTalker* são responsáveis pela entrada e saída de membros, respectivamente. Estas operações não foram modificadas na camada TF, e por isso não as apresentaremos em detalhes. Essas operações são chamadas pela camada superior da mesma forma em quase todas as camadas, como mostra a figura 4.8. Apenas a camada REACH obtém as informações de entrada e saída de novos membros, armazena essa informação no evento Visão (*evView*) e chama a operação *upHandleEvent* para enviar o evento Visão às camadas superiores.

```
public synchronized void dnRegisterTalker(iBusURL channel)
      throws AlreadyRegistered, CommException
      {
          // pass this event down the stack:
          if (below() != null) below().dnRegisterTalker(channel);
      }
```

Figura 4.11: Operação *dnRegisterTalker*

As operaçãoes de *dnSubscribe*, *dnUnsubscribe* e dn*UnregisterTalker* possuem a mesma sintaxe da figura 4.11.

4.3.2 – Classe *BlockMatrix*

Nessa classe encontram-se as operações responsáveis pelo gerenciamento de matriz de blocos: criação da matriz e inserção e remoção de mensagens. A Matriz de Blocos é organizada em uma estrutura de dados com dois níveis: o primeiro nível é uma tabela

hash endereçada pelos números de blocos das mensagens e o segundo nível é uma lista linear onde são armazenadas as mensagens [MAC94].

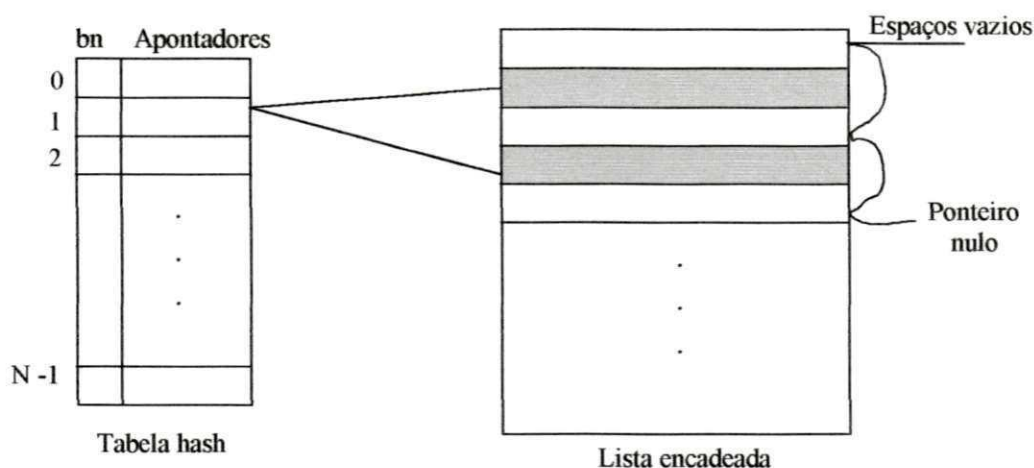


Figura 4.12: Estrutura da Matriz de Blocos

A figura 4.12 apresenta a estrutura da Matriz de Blocos. Cada entrada da tabela *hash* contém um número de bloco e aponta para as mensagens associadas com este número de bloco, armazenadas na lista encadeada. A matriz é criada recebendo como parâmetro o tamanho do grupo. As mensagens são inseridas na matriz através da operação *insert(MessageEvent msg, int bn)*, recebendo como parâmetros a mensagem e o número de bloco da mensagem. As mensagens são localizadas através das operações *fisrtMessage* e *nextMessage*, sendo então entrgues a aplicação. A operação *stabilise* remove as mensagens estáveis da matriz de blocos (figura 4.13).

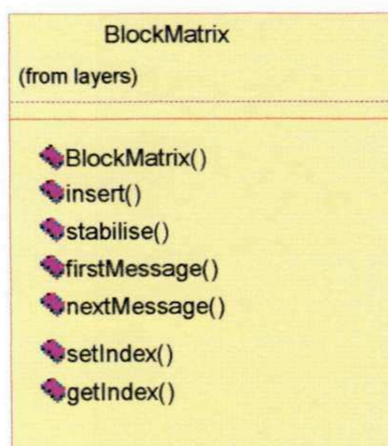


Figura 4.13: Operações da classe *BlockMatrix*

4.3.3 – Classe *TimeOut*

A classe *TimeOut* é a classe base para a *thread* LTS. A cada número de bloco criado na Matriz de Blocos, é iniciado um *TimeOut* correspondente. Se o objeto membro do grupo não enviar mensagens durante um intervalo de tempo definido em *interval*, o *TimeOut* despertará o LTS, que deverá enviar mensagens nulas. Uma vez que um objeto *TimeOut* é criado, operações de *setTimeOut*, *TimeOutCancel*, *FirstTimeOut* (retorna o *TimeOut* mais "antigo") e *NextTimeOut* (cancela o *TimeOut* mais "antigo"), podem ser invocadas pela *thread* LTS (figura 4.14).

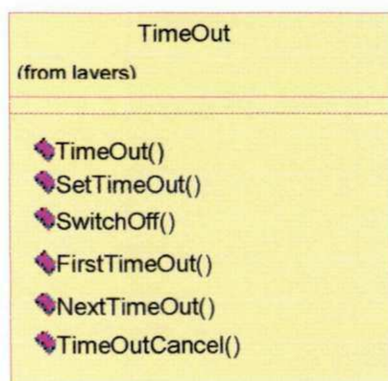


Figura 4.14: operações da classe *TimeOut*

A operação de *SetTimeOut* incia um *TimeOut*. Esta operação é chamada pelo *upHandleEvent* da classe TF quando um novo *bn* é criado na Matriz de Blocos. A operação *TimeOutCancel* (cancela todos os *TimeOut*) pode ser soliciatada tanto pela operação *dnPush* da camada TF após o recebimento de uma mensagem enviada pela aplicação, como pela *thread* LTS, após o envio de uma mensagem nula, para cancelar todos os *TimeOut* anteriores. As outras operações, *FirstTimeOut* (retorna o *TimeOut* mais "antigo"), *NextTimeOut* (cancela o *TimeOut* mais "antigo"), e *SwitchOff*, são chamadas pela *thread* LTS

4.3.4 – *Thread* LTS

A *thread* LTS implementa as operações para o envio de mensagens nulas após uma mensagem recebida de *TimeOut*. A mensagem enviada será do tipo *MessageNull*, e quando recebida pela operação *upHandleEvent*, atualizará as estruturas de dados. Mensagens nulas não são entregues à aplicação.

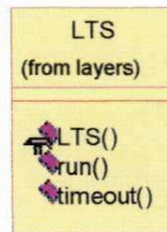


Figura 4.15: Operações da classe LTS

4.3.5 – *Thread Delivery*

A *thread Delivery* remove as mensagens da Matriz de Blocos e as envia à camada superior STACK. As mensagens só serão entregues quando os blocos estiverem completos. Para isso, essa *thread* compara em intervalos de tempos predefinidos, o vetor LRV. Se há blocos completos, então as mensagens são removidas e entregues à aplicação. Durante a mudança de visão, a *thread Delivery* interrompe a entrega de mensagens à camada STACK. Este procedimento é necessário para garantir a atomicidade na entrega de mensagens aos membros dos grupos.

Além das *threads* LTS e Delivery, outras *threads* compõem o iBusTF, localizadas nas camadas REACH e IPMCAST, como ilustra a figura 4.16.

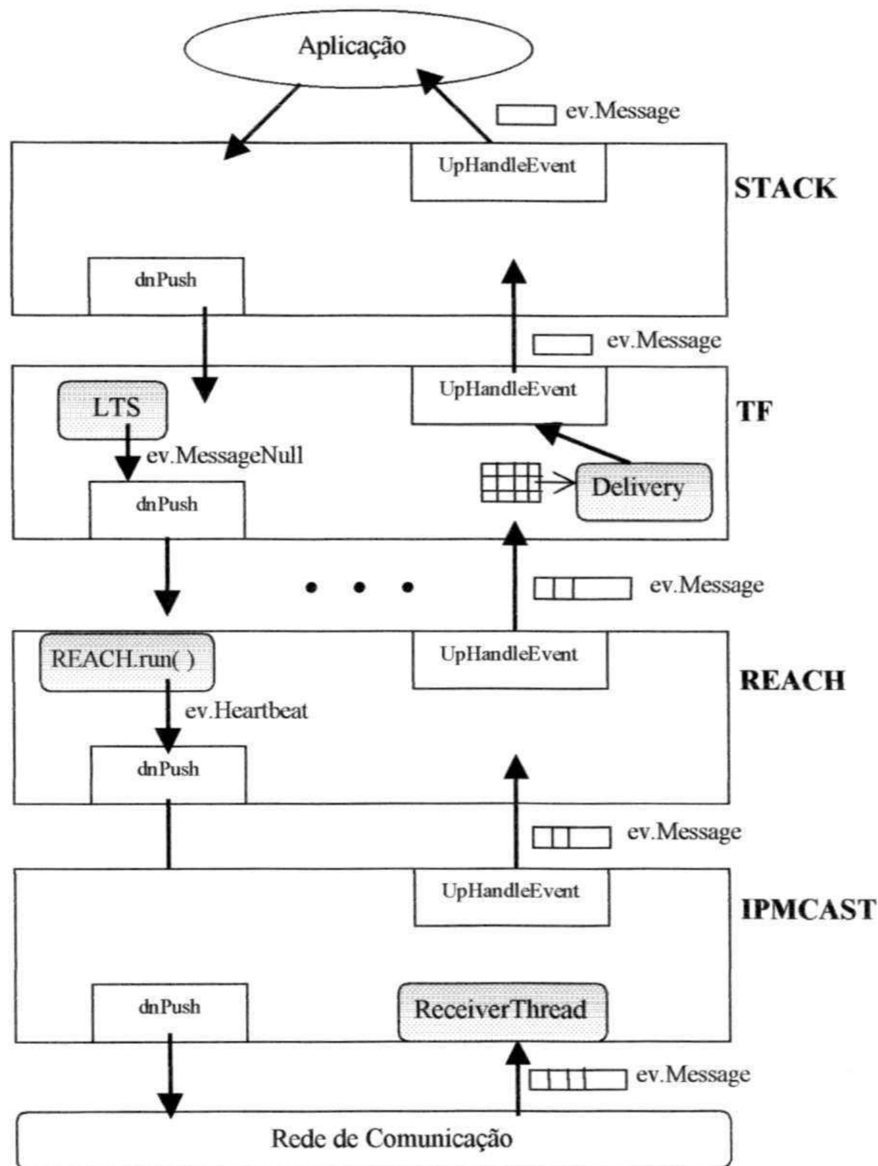


Figura 4.16: *Threads* do iBusTF

Uma mensagem trafega pela rede sendo recebida na camada IPMCAST. Nesta camada, a thread *ReceiverThread* fica bloquedada esperando por novas mensagens no canal de comunicação. Após a recepção da mensagem, a camada IPMCAST retira do cabeçalho da mensagem as informações correspondentes a sua camada e envia a mensagem para a camada REACH. A camada REACH não remove nenhuma informação da mensagem,

apenas a envia para a camada acima. A camada REACH mantém uma *thread* responsável por enviar sinais (*heartbeats*) periódicos para os grupos em que o objeto aplicação está inscrito.

As camadas NAK, FIFO e FRAG, localizadas entre as camadas REACH e TF, não possuem *threads*. Na camada TF a mensagem é armazenada na Matriz de Blocos. A *thread Delivery* verifica se há blocos completos na Matriz de Blocos, remove as mensagens da Matriz e as envia para a camada STACK. A *thread* LTS é responsável por enviar , quando necessário, mensagens nulas para completar blocos. Finalmente a mensagem é enviada da camada STACK para a aplicação.

4.3.6 – Classe *HandleMembers*

Esta classe atribui um identificador único (*objectID*) aos membros dos grupos. Este identificador é necessário para localizar os objetos nas estruturas de dados. O *objectID* é obtido partir do endereço IP onde o objeto está em execução, e a porta pela qual os eventos são recebidos, para cada membro do grupo. Por exemplo, digamos um grupo formado por três objetos com os seguintes endereços:

objeto O_1: ibus://200.17.144.91:2020
objeto O_2: ibus://200.17.144.91:2021
objeto O_3: ibus://200.17.144.93:2030

Estes endereços são obtidos depois que os objetos se registram em um grupo, enviados pela camada REACH. Cada objeto utiliza uma porta diferente. Na camada TF estes endereços são convertidos para o tipo *long* e armazenados em ordem crescente numa tabela. O índice da tabela corresponde ao identificador único de cada membro, e será utilizado para identificar os membros nas estruturas de dados. Cada membro do grupo mantém uma tabela de endereços.

Este procedimento foi necessário para que cada objeto membro de um grupo recebesse um identificador único e fosse conhecido por todos os outros membros. Utilizando o

endereço IP dos três objetos anteriores, por exemplo, temos que após a conversão desse valor em um long e o armazenamento em uma tabela em ordem crescente, objeto O_1 recebeu identificador 1, o objeto O_2 identificador 3 e objeto O_3 identificador 2. Todos os objetos executam este procedimento, obtendo os mesmos valores, e garantindo a consistência nos vetores e matriz de blocos utilizados.

Nessa dissertação, não tratamos a inserção de membros após a formação inicial da Matriz de Blocos.

4.3.7 – Classe *MessageNull*

Essa classe é responsável pela criação dos eventos para envio de mensagens nulas utilizadas para completar blocos. Todos os eventos criados pelas camadas são subclasses da classe abstrata *iBus.Event*. A classe *MessageNull* contém as operações para a criação das mensagens nulas e a adição e remoção de dados no cabeçalho da mensagem nula.

Figura 4.17: Operações da classe *MessageNull*

As operações da classe *MessageNull* são apresentadas na figura 4.17. A operação *id* retorna o identificador desta operação. *setData* é utilizada para armazenar as informações que serão enviadas neste evento. A operação *getID* retorna o identificador do objeto que enviou este evento e *getBn* retorna o *bn* do evento de mensagem nula.

4.3.8 – Classe *Suspect*

A classe *Suspect* é responsável pela criação dos eventos de suspeita. O evento de suspeita (*ev.Suspect*) armazena o identificador (*SuspID*) do emissor, o identificador (*SuspIDs*) do membro suspeito de falha e o número de blocos da última mensagem recebida do membro suspeito de falha (*Bn*) através da operação *setData(int suspID, int SuspIDs, int Bn)*. Esta operação é chamada após o recebimento de um evento de visão (evView) enviado pela camda REACH informando a falha de algum membro.

As operaçãos *getSuspID* (retorna o emissor do evento), *getSuspIDs* (retorna o identificador do membro suspeito de falha) e *getBn* (retorna o número de bloco da última mensagem recebida pelo membro suspeito), são chamadas através da operação *upHandleEvent* da classe TF após o recebimento de um evento de Suspeita enviado por um dos membros do grupo.

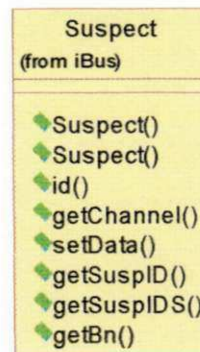


Figura 4.18: Operações da classe *Suspect*

4.3.9 – Classe *Refute*

A classe Refute é responsável pela criação dos eventos de negação do evento de suspeita. O evento Refute armazena o identificador do emissor do evento (*RefID*), o identificador do membro suspeito de falha (*RefIDS*) e o *bn* da última mensagem recebida por este membro suspeito (*BN*), que deverá ser maior que o *bn* recebido na mensagem de suspeita. A classe *Refute* contém as mesmas operações da classe *Suspect*.

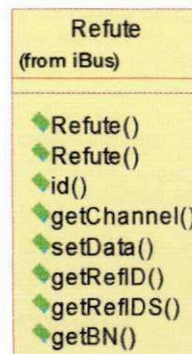


Figura 4.19: Operações da classe Refute

As operaçãos *getRefID* (retorna o emissor do evento), *getRefIDs* (retorna o identificador do membro suspeito e cuja falha não foi confirmada) e *getBn* (retorna o número de bloco da última mensagem recebida por este membro), são chamadas através da operação *upHandleEvent* da classe TF após o recebimento de um evento de *Refute* enviado por um dos membros do grupo.

4.3.10 – Classe *Recovery*

A classe *Recovery* é responsável pela criação dos eventos de "recuperação" ou seja, evento de recuperação das mensagens não recebidas do membro suspeito. O evento *Recovery* armazena o *objectID* do emissor, o *objectID* do membro de quem se deseja receber as mensagens e o número de bloco a partir de onde se deseja receber as mensagens. A classe *Recovery* também contém as mesmas operações da classe *Suspect* e *Refute*.

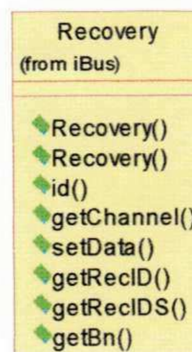


Figura 4.20: Operações da classe *Recovery*

4.3.11 – Classe *Confirmed*

A classe *Confirmed* é responsável pela criação dos eventos de confirmação. O evento *Confirmed* armazena o *objectID* do emissor, e o vetor dos membros ditos "falhos" após a realização do acordo. A classe *Confirmed* contém as mesmas operações da classe *Suspect*, *Refute* e *Recovery*.

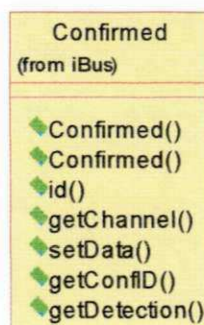


Figura 4.21: Operações da classe *Confirmed*

4.3.12 – Classe *Semaphore*

Outro recurso utilizado nesta implementação foi o uso de semáforos, visto que as informações armazenadas nas estruturas de dados (vetores, matrizes) são compartilhadas. Utilizamos os semáforos para garantir que diferentes objetos da camada TF, incluindo as *threads*, tenham acesso exclusivo a essas estruturas. Quando um objeto vai executar algum processamento em uma das estruturas compartilhadas, ele chama a operação *down* que por sua vez chama a função *wait( )* do Java bloqueia o uso desta estrutura e apenas quando a estrutura for liberada, através da operação *up*, que chama a função *notify( )* do Java, os outros objetos que estão esperando em uma fila poderão utilizar a estrutura.

O iBusTF utiliza a função *synchronized* do Java que garante acesso exclusivo às operações. Esta função é utilizada na operação *dnPush* da classe TF para garantir que enquanto algumas estruturas estão sendo modificadas, e uma mensagem chegar através do operação *upHandleEvent*, esta só tenha acesso as estruturas quando a operação

dnPush não as estiver mais utilizando. No entanto, na camada TF utilizamos *threads* que também compartilham as estrutras de dados com essas duas operações. Se adicionassemos outra função *synchronized* à operação *upHandleEvent* poderia ocasionar um *deadlock*. A solução adotada por nós nessa implementação, foi então a criação da classe *Semaphore*.

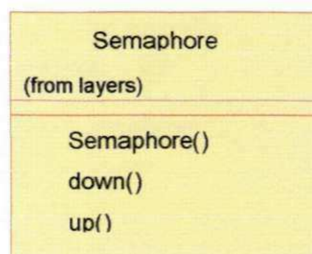


Figura 4.22: Operações da classe Semaphore

4.4 – Avaliação de Desempenho

Os objetivos principais desta avaliação de desempenho foram: primeiro, observar a consistência dos membros dos grupos com a ordenação total de mensagens, e segundo, observar o comportamento do iBusTF na presença de falha de objetos membros do grupo. No primeiro experimento realizado variamos o grupo de 2 a 6 membros, onde cada membro enviava e recebia um número de mensagens variando de 100 a 1000, e onde as mensagens recebidas eram armazenadas em um arquivo. Após o experimento, através de uma função de comparação do Unix, observamos a ordenação das mensagens de cada objeto membro do grupo.

Além de comparar os arquivos dos objetos (não achamos necessário a apresentação dos arquivos na dissertação), utilizamos um outro mecanismo de medição, o mecanismo, *Round Trip*, que calcula o tempo que um objeto leva para enviar uma mensagem a um grupo e receber a resposta de confirmação (*ack*) de um dos membros do grupo.

No segundo experimento realizado forçamos a parada de um membro por vez e observamos o comportamento dos objetos membros do grupo, desde a suspeita de falha de um dos membros, até a sua completa remoção do grupo. Neste experimento,

calculamos o tempo para realização do acordo sobre a falha de um membro. Os testes foram realizados utilizando-se quatro PCs 300Mhz, conectados por uma rede *Ethernet* 10Mbps, sistema operacional Windows 95 e JDK1.2.2.

No mecanismo de *Round Trip*, um objeto aplicação denominado *clientRT* envia mensagens a um grupo de aplicações servidoras e recebe de volta a confirmação do recebimento das mensagens. Depois que a primeira confirmação é recebida, uma nova mensagem é enviada ao grupo. O tamanho do grupo variou de 2 a 6 objetos e para cada tamanho de grupo foram enviadas 100 mensagens. Distribuímos os objetos ao longo das quatro estações utilizadas para os testes, e definimos o valor do LTS em 150ms (*milissegundos*). Este valor representa o tempo que a aplicação servidora espera antes de verificar se precisa enviar mensagens nulas. Este procedimento é fundamental para garantir a entrega das mensagens à aplicação, pois como dito anteriormente, as mensagens são apenas retiradas da matriz de blocos e entregues à aplicação quando os blocos estão completos.

Este experimento foi realizado com o intuito de medir o *overhead* causado pelos mecanismos de TF. Esse experimento foi realizado tanto no ambiente original iBus como no ambiente Tolerante a Falhas iBusTF implementado por nós. O diagrama de sequência do *clientTimeRT* é apresentado abaixo.

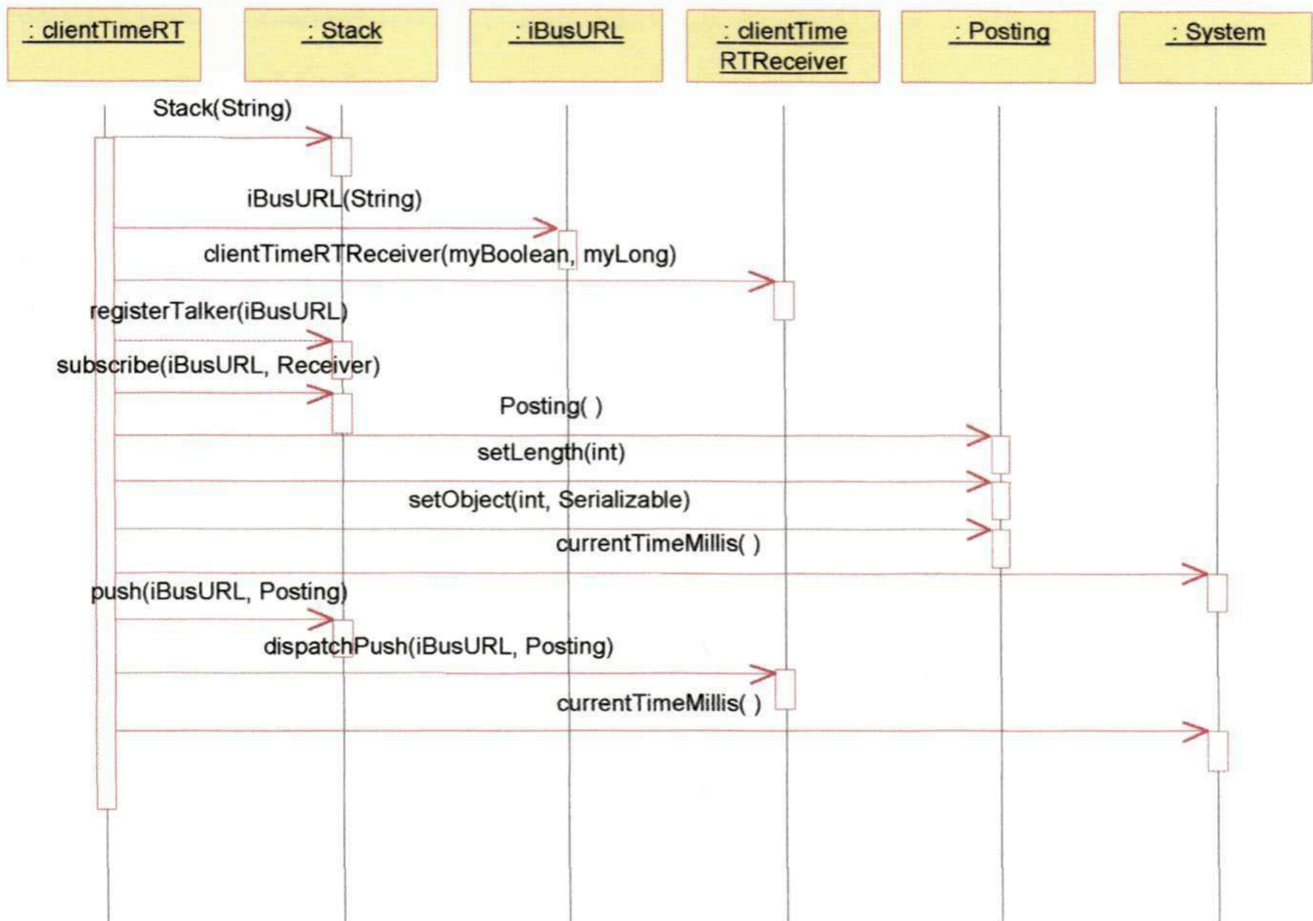


Figura 4.23: Diagrama de Sequência do *clientTimeRT*

A figura 4.23 apresenta o diagrama de sequência da aplicação *clientTimeRT*. Inicialmente a aplicação cria a pilha de camadas (operação *Stack*) e depois define o grupo ao qual irá fazer parte (operação *iBusURL*) na classe iBusURL. A classe *clientTimeRTReceiver* define as operações para recebimento das mensagens enviadas pelo grupo. O *clientTimeRT* se registra no grupo através das operações *registerTalker* e *susbcribe*. A operação *Posting* cria um objeto que irá aemazenar as mensagens enviadas ao grupo. A operação de *setLength* define quantos objetos serão armazenado e *setObject* insere os objetos no Posting. Antes de enviar a mensagem, o *clientTimeRT* guarda o tempo da máquina em milissegundos(*currentTimeMillis*). O Posting é então enviado ao grupo (operação *push*). A operação *dispatchPush* recebe as mensagems enviadas ao grupo e as mostra na tela. Novamente o valor da máquina é guardado e a diferença entre

o valor incial e o valor final é armazenado em um arquivo. Após este procedimento o *clientTimeRT* envia outra mensagem ao grupo.

O gráfico abaixo mostra a média obtida através do mecanismo de *Round Trip* no iBusTF em relação a variação do tamanho do grupo.

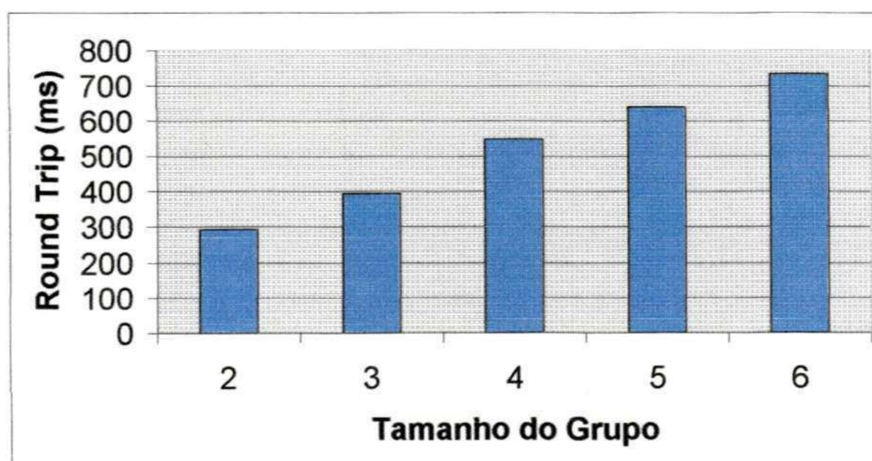


Figura 4.24: Escalabilidade segundo *Round Trip* no iBusTF.

Observamos a partir do gráfico da figura 4.24 que à medida que o tamanho do grupo aumenta, o tempo de *Round Trip* também aumenta. O gráfico da figura 4.25 apresenta os resultados obtidos (em *milissegundos*), a partir do mesmo experimento de *Round Trip* realizado no iBus.

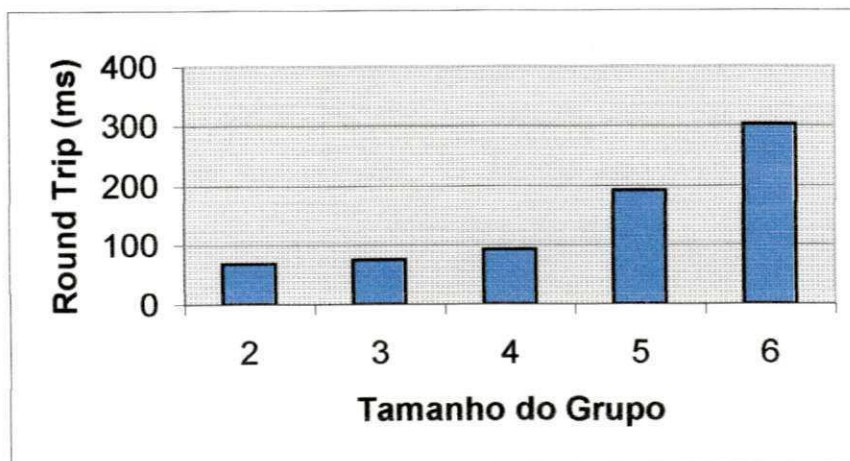


Figura 4.25: Escalabilidade segundo *Round Trip* no iBus.

Como esperado, os valores obtidos no iBusTF são maiores que no iBus, visto que o iBusTF introduz um atraso para entrega das mensagens à aplicação. Este atraso é necessário para garantir a ordenação total das mensagens entregues à aplicação. Uma das maneiras de melhorar o *overhead* do iBusTF é diminuir o valor de LTS. Outros teste foram realizados com o valor de LTS variando entre 50ms e 100ms, onde foi observada um descrécimo no atraso para a entrega das mensagens à aplicação.

O gráfico da figura 4.26 mostra o tempo para realização do acordo sobre a suspeita de falhas dos membros. O tempo médio para realização do acordo é a diferença entre o recebimento de uma mensagem de visão enviada pela camada REACH do iBus, com a suspeita de falha de um membro, e a sua completa remoção do grupo pelo iBusTF. Este cálculo é importante visto que durante a realização do acordo sobre as falhas, a *thread Delivery* bloqueia a entrega de mensagens à aplicação, aumentado o "atraso para entrega das mensagens". Este atraso é a diferença entre a recepção de uma mensagem pela camada mais inferior da pilha de camadas do iBus (camada IPMCAST) e a entrega dessa mensagem à aplicação.

Neste experimento, foram realizados testes com grupos de 2 a 6 objetos, distribuídos nas quatro estações de trabalho, e onde apenas um objeto falha durante cada execução. As falhas são introduzidas através de uma parada, isto é, forçamos a parada de um membro do grupo por vez. O tempo médio para realização do acordo (em *milissegundos*) é representado no gráfico abaixo.

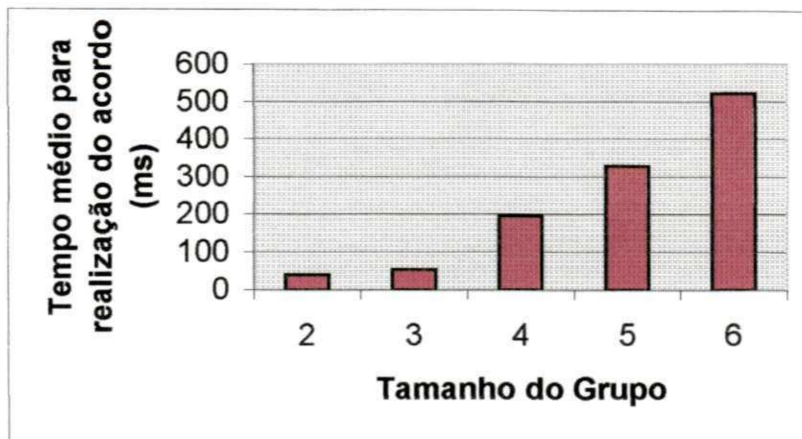


Figura 4.26: Gráfico do tempo médio para realização do acordo sobre a suspeita de falhas.

O tempo obtido é o resultado da média do acordo realizado por todos os objetos. Como observado no gráfico da figura 4.26, o tempo médio para realização do acordo também aumentou com a adição de novos membros ao grupo.

4.5 - Conclusão

Nesse capítulo, apresentamos uma abordagem para introduzir tolerância a falhas às aplicações Java através do ambiente iBusTF. O iBusTF é uma extensão ao sistema de Comunicação em Grupo iBus escrito puramente em Java. O iBusTF estendeu o iBus com uma nova camada, denominada TF, onde foram introduzidos os protocolos de Ordenação Total e *Membership* Atômico baseados no modelo proposto na BCG.

O iBusTF, assim como o iBus, permite a criação da pilha de camadas de protocolos em tempo de execução. No iBusTF as mensagens são entregues com ordem total e os membros suspeitos de falha são removidos do grupo após um acordo sobre a falha dos mesmos. Apenas quando todos os membros operacionais do grupo concordam sobre a suspeita de falha de um determinado membro, este membro é removido do grupo.

Realizamos experimentos para avaliar o desempenho do ambiente iBusTF através dos mecanismos de *Round Trip* e do tempo médio para realização do acordo sobre a falha

de membros. No mecanismo de *Round Trip* um cliente envia mensagens a um grupo de objetos servidores e só após o recebimento da confirmação (*ack*) de um dos servidores, este envia uma nova mensagem ao grupo. Observamos nesse experimento que o tempo obtido no iBusTF é maior que no iBus, justificado pelo atraso introduzido no iBusTF para garantir que as mensagens sejam entregues à aplicação em ordem total.

O tempo médio para realização do acordo sobre a suspeita de falha de um membro, entre os membros operacionais de um grupo, é a diferença entre a o envio ao grupo de uma suspeita de falha de um membro e a sua remoção do grupo. Para cada tamanho de grupo, os grupos também variaram de 2 a 6 membros e forçamos a falha de um membro por vez através da parada de um objeto (falha do tipo *crash*). Durante a realização do acordo sobre as suspeitas de falhas, a *thread Delivery* interrompe a entrega de mensagens à aplicação, o que também contribui para o atraso introduzido no iBusTF. O nosso objetivo principal nesses experimentos, contudo, não era comparar a performance do iBusTF em relação ao iBus, mas observar e testar o comportamento do iBusTF na presença de falhas de membros dos grupos.

Os resultados obtidos mostram o *overhead* necessário à introdução de garantias de Tolerância a Falhas num ambiente Java dentro do modelo proposto nessa dissertação. Houve um acréscimo no tempo, de 360 ms (milissegundos) em média, entre o intervalo de envio e entrega das mensagens à aplicação no iBusTF em relação ao ambiente original iBus. Todavia, esse *overhead* não é muito significativo se comparado com as garantias oferecidas em um ambiente Tolerante a Falhas, e que em certas aplicações, principalmente em aplicações críticas como controle de tráfego aéreo e transações bancárias, é fundamental a confiabilidade das aplicações.

Capítulo 5

Comparação do iBusTF com outras propostas para Tolerância a Falhas em Java

Este capítulo apresenta outras trabalhos para introduzir Tolerância a Falhas a Java: o Filterfresh, o ROAPI e o JavaGroups, e compara os principais aspectos dessas propostas em relação ao iBusTF.

5.1 – Introdução

Alguns trabalhos para Tolerância a Falhas em Java, foram encontrados na literatura. Apesar de algumas trabalhos em andamento, ainda não há um número significativo de projetos nessa área se comparado ao crescimento no número de aplicações distribuídas em desenvolvmento utilizando a linguagem Java.

As linhas de pesquisa se diferenciam na forma de integrar Tolerância a Falhas em Java. O *Filterfresh*[CHR+99] por exemplo, propõe adicionar Tolerância a Falhas aos objetos servidores e ao registrador do RMI, o *JavaGroups*[BAN98], um ambiente similar ao iBus, é uma ferramenta de Comunicação em Grupo escrita totalmente em Java, e o *ROAPI*[LEW99] é uma API para a construção de objetos replicados em Java. Esses ambientes são comparados em relação a forma de introduzir Tolerância a Falhas a Java Ao final do capítulo comparamos as principais características destes ambientes, como portabilidade e desempenho, em relação ao iBus. Esses ambientes são apresentados a seguir.

5.2 – Filterfresh

O Filterfresh[CHR+99] é uma ferramenta para a construção de aplicações Java Tolerantes a Falhas baseada no *Remote Method Invocation* (RMI). O objetivo do Filterfresh é adicionar tolerância a falhas aos objetos servidores e ao Registrador do RMI.

Como visto na seção 3.2.2, uma aplicação RMI consiste de um objeto cliente e um objeto servidor. O objeto cliente requisita um método no objeto servidor através de sua referência. A referência dos objetos servidores são armazenadas nos Registradores e a ação de registro do servidor no Registrador é denominada *bind*. O objeto cliente envia uma *string* como parâmetro ao Registrador e obtém a referência do objeto servidor. Essa operação é denominada *lookup*.

Sob a ótica de Tolerância a Falhas, tanto o cliente quanto o Registrador são pontos de falhas no RMI. Na ocorrência de alguma falha no objeto servidor, ao tentar invocar um método nesse servidor, o cliente recebe como resposta uma mensagem de erro. Em se tratando do Registrador, se este não estiver operacional, o cliente ficará impossibilitado de obter as referências dos objetos servidores disponíveis. Mesmo que os objetos servidores estejam operacionais, o cliente não conseguirá invocá-los.

O Filterfresh adiciona Tolerância a Falhas aos objetos servidores e ao Registrador através de replicação. A replicação no Filterfresh assegura que se um objeto servidor falhar, a solicitação requisitada pelo cliente será enviada e executada por outro objeto servidor, de forma transparente para o cliente. Essa transparência é conseguida através do mecanismo FT Unicast (*Transparent Client-Side Fault Tolerance*), que será apresentado posteriormente. Da mesma forma que os objetos servidores, o Registrador também é replicado. Se um Registrador falhar, o cliente poderá obter as referências dos objetos servidores em outro Registrador.

O componente principal no Filterfresh é o Gerenciador de Grupos (*GroupManager*). O Gerenciador de Grupos é responsável por coordenar os grupos (entrada, saída e detecção de membros falhos) e por manter os membros dos grupos consistentes através de difusão de mensagens (*multicast* confiável). O Gerenciador de Grupos é local para cada objeto replicado e os gerenciadores locais juntos formam um grupo. Replicação passiva com a presença de um coordenador. O Gerenciador de Grupos é responsável pelas seguintes funções:

1. **Criação do Grupo:** Quando o primeiro objeto é inicializado, o Gerenciador de Grupos desse objeto define um grupo formado por apenas este objeto através da operação *createNewGroup*( ). O primeiro objeto em um grupo é sempre o coordenador do grupo.

2. **Entrada de Membros:** Um objeto que deseja entrar em um grupo que já existe invoca a operação *joinExistingGroup( )*. Essa solicitação para entrada no grupo é enviada para os outros objetos membros dos grupos, pelo coordenador, para que estes atualizem sua visão do grupo. Depois que os objetos membros do grupo recebem a solicitação, o estado de um dos objetos é transferido para o novo membro. Esta atualização é possível porque o *Filterfresh* utiliza replicação, onde um coordenador é responsável por gerenciar os eventos do grupo.

3. **Saída de Membros:** Um membro pode sair de um grupo através de uma chamada à operação *leave( )*. Da mesma forma que a operação *joinExistingGroup( )* a solicitação para saída de um grupo é enviada a todos os membros operacionais do grupo para que esses atualizam suas visões do grupo.

4. ***Multicast* confiável:** As mensagens enviadas a um grupo atendem a duas propriedades: atomicidade e ordenação total. Essas garantias são necessárias para manter os membros dos grupos consistentes. Para atender a esse requisito, é utilizado um membro coordenador em cada grupo. Por exemplo, um objeto que deseja enviar uma mensagem para o grupo, armazena primeiramente essa mensagem em seu *buffer* local e invoca a operação *multicast( )*. A mensagem é então enviada ao coordenador do grupo. O objeto emissor permanece bloqueado até receber uma mensagem de confirmação do coordenador. Se o objeto emissor não receber uma mensagem de confirmação do coordenador em um tempo preestabelecido, ele reenvia a mensagem. O coordenador ao receber a mensagem do objeto emissor, incrementa o número de seqüência dessa mensagem e a envia aos outros membros do grupo. Após receber confirmação de todos os membros do grupo, o coordenador notifica o objeto emissor que a operação foi realizada com êxito.

5. **Detecção de Falhas:** O mecanismo de detecção de falhas é baseado em *timeouts*. O Gerenciador de Grupos de cada membro envia sinais periódicos ao grupo e se um membro permanece por um tempo predefinido sem enviar sinais, ele será suspeito de

falha. Quando um membro é considerado falho, uma nova visão é gerada excluindo esse membro do grupo. A visão de um grupo no Filterfresh é formada pelo identificador de cada membro operacional do grupo e um número que identifica a visão. Esse número é incrementado a cada mudança de visão, sendo utilizado para assegurar que uma mensagem do membro que falhou endereçada ao grupo na visão anterior, não será aceita pelos membros que permaneceram no grupo após instalada a nova visão. O protocolo de mudança de visão é dividido em duas etapas: na primeira etapa são determinados os membros operacionais e não suspeitos de falha, e na segunda etapa é instalada a nova visão, formada pelos membros pertencentes à primeira etapa.

Como descrito anteriormente, o objetivo principal do Gerenciador de Grupos é adicionar Tolerância a Falhas aos objetos servidores e ao Registrador do RMI através de replicação. A figura 5.1 mostra como o Gerenciador de Grupos é utilizado no Registrador Tolerante a Falhas do *Filterfresh* (*FT Registry*).

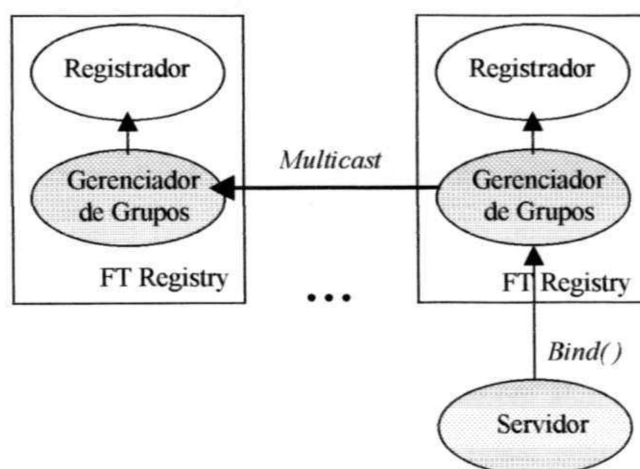


Figura 5.1: Operação de atualização no *FT Registry*

Um objeto servidor se registra através da operação *bind( )* em um dos Registradores, e esta atualização é propagada para os outros registradores replicados pelo Gerenciador de Grupos através de *multicast* confiável. O cliente pode então obter a referência de um objeto

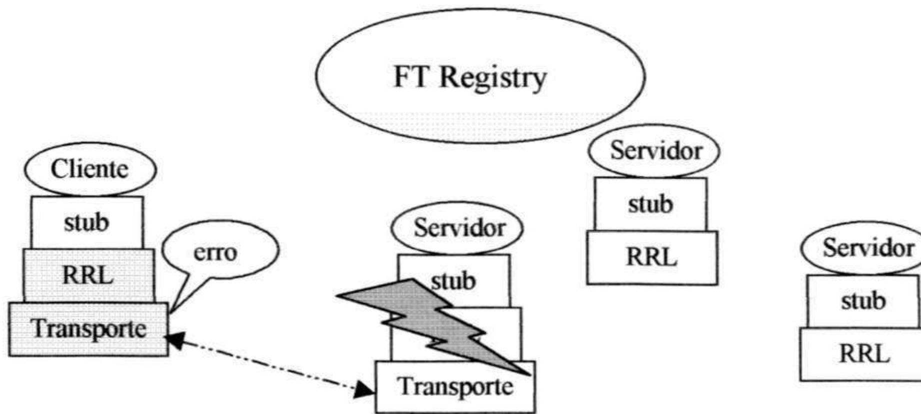


Figura 5.3: Mensagem de erro de um objeto servidor interceptada na camada RRL

Esse problema é resolvido no Filterfresh, interceptando na camada RRL a mensagem de erro que seria retornada ao cliente (figura 5.3). A camada RRL executa a operação *reverse lookup*( ) em qualquer um dos registradores replicados utilizando a mesma referência (figura 5.4).

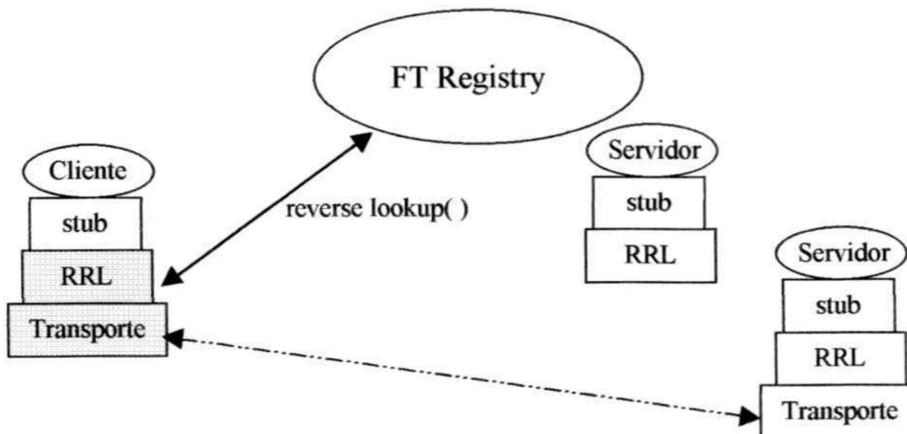


Figura 5.4: Chamada a um objeto servidor replicado

O registrador envia o nome do objeto (*String*) que será utilizado para executar uma nova operação de *lookup*( ). A operação *lookup*( ) irá retornar a referência de uma das réplicas

desse objeto servidor. O objeto servidor replicado será então invocado e o resultado enviado de volta ao cliente.

5.3 – ROAPI

ROAPI (*Replicated Object API*) [LEW99] é formado por um conjunto de classes Java que permitem a construção de aplicações distribuídas estruturadas em grupos, como aplicações *groupware*, aplicativos eletrônicos em rede e aplicações de ensino à distância. O objetivo principal do ROAPI é oferecer uma API que permita a construção de objetos Java replicados, ou seja, o ROAPI permite a replicação de objetos Java e a estruturação dos mesmos em grupos.

Os grupos são dinâmicos, novos objetos podem ser adicionados e removidos dos grupos. O ROAPI permite que o programador da aplicação tenha controle sobre a replicação dos objetos. Os objetos replicados formam o "Espaço de Replicação", como mostra a figura abaixo.

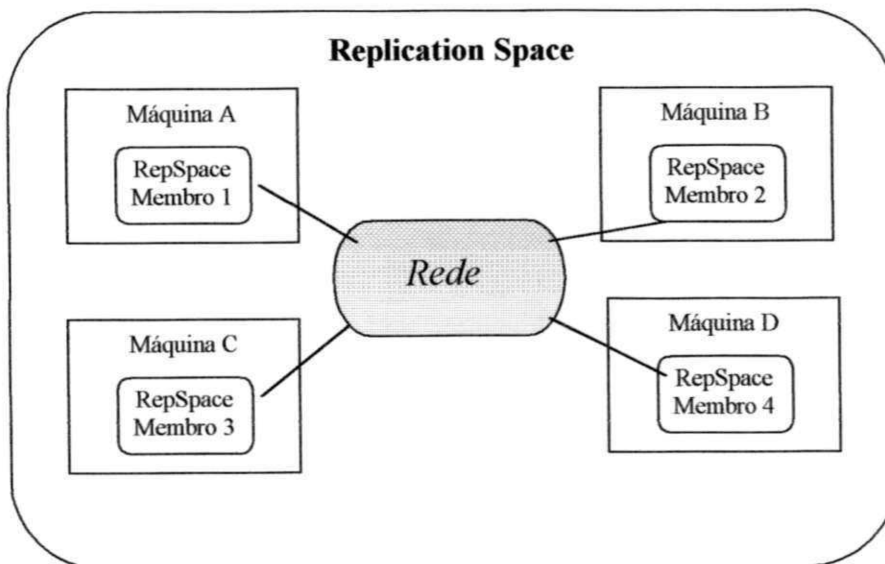


Figura 5.5: Estrutura de um de um grupo no ROAPI

A figura 5.5 [LEW99] mostra a estrutura de um grupo no ROAPI. Os objetos membros dos grupos são instâncias da classe *RepSpace* e juntos formam o Espaço de Replicação (*Replication Space*). A comunicação entre os membros pode ser realizada através de *multicast* confiável. As mensagens são enviadas com garantias de ordenação total e atomicidade. O ROAPI também implementa um protocolo de *membership* que mantém atualizados os membros dos grupos da entrada, saída ou falha de membros. Os protocolos de *multicast*, ordenação atomicidade e *membership* do ROAPI são baseados na primitiva de comunicação *Dynamic Terminating Multicast* [GS94].

Além de *multicast* confiável, a comunicação no ROAPI também pode ser realizada através do protocolo TCP/IP, da classe Java *MulticastSocket*, do RMI e também utilizando o iBus. A classe *iBusRepSpace* define uma interface para que o ROAPI utilize o iBus como camada de transporte.

O ROAPI encontra-se em fase inicial de desenvolvimento.

5.4 – JavaGroups

JavaGroups[BAN98, BAN99], em desenvolvimento na Universidade de Cornell, é um ambiente para a construção de aplicações Java tolerantes a falhas baseado nos sistemas Horus[RMB96] e Ensemble[HAY98]. A principal característica do JavaGroups em relação a outros Sistemas de Comunicação em Grupo é a presença de classes Java localizadas abaixo do nível da aplicação, utilizadas para facilitar o trabalho do programador da aplicação. Estas classes realizam tarefas como tranferência de estado, ou chamada síncrona sobre sistemas assíncronos.

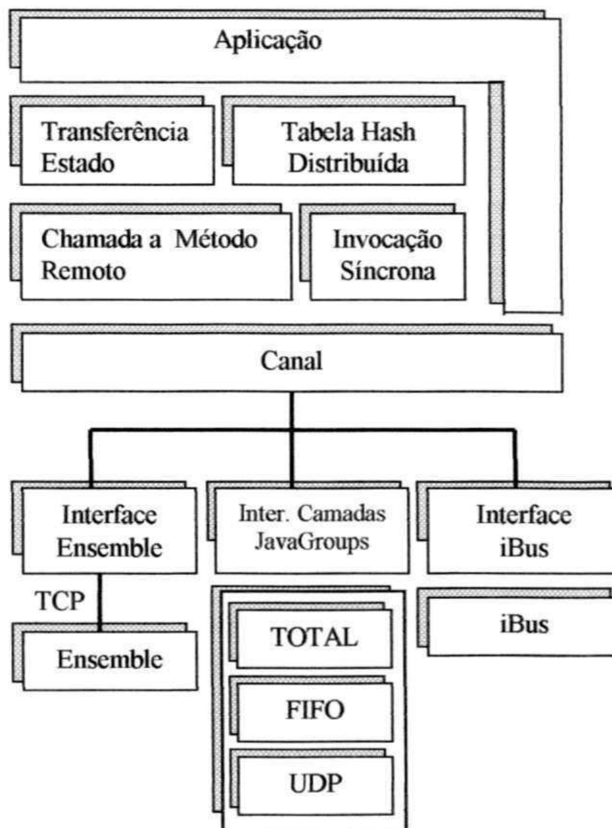


Figura 5.6: Arquitetura do JavaGroups

O JavaGroups possui uma arquitetura estruturada em camadas. O ponto central da arquitetura do JavaGroups é o conceito de Canal. Cada canal tem um nome e canais com o mesmo nome formam um grupo. O Canal oferece as funcionalidades básicas para o gerenciamento dos grupos: envio e recebimento de mensagens pelos membros de um grupo e notificação dos membros que entraram e saíram dos grupos. Um objeto para entrar em um grupo deve primeiro criar um canal e se "conectar" a este, e para sair do grupo deve se "desconectar" do canal (figura 5.7).

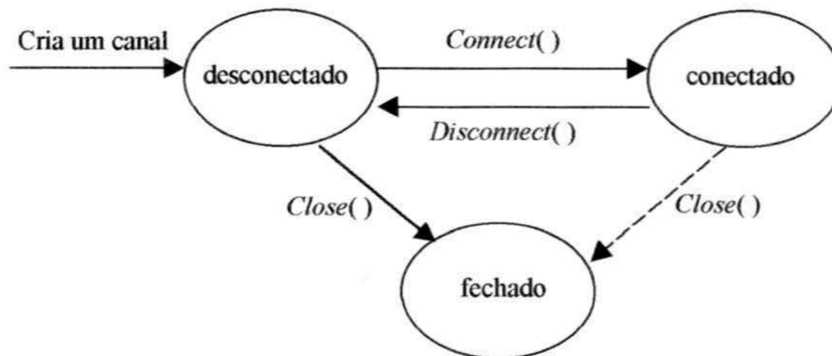


Figura 5.7: Estados de um canal

Quando um canal é inicialmente criado por um objeto, ele se encontra no estado "desconectado". Após a operação *Connect( )* o objeto passa a fazer parte do grupo e pode enviar e receber mensagens do grupo. Uma mensagem enviada a um canal é recebida por todos os outros canais com o mesmo nome. O objeto sai do grupo através da operação *Disconnect( )*. Após essa operação o Canal retorna ao estado inicial, podendo se "reconectar" posteriormente. Um canal conectado ou "desconectado" pode passar para o estado de fechado. Quando um canal transita do estado conectado para o estado fechado, o canal é primeiramente "desconectado" e depois fechado (figura 5.7). O Canal no estado fechado fica impossibilitado de realizar qualquer operação (enviar, receber mensagens, etc.) e este será removido posteriormente pelo mecanismo de *Garbage Collection* do Java[ECK98].

Entre a Camada de Aplicação e o Canal localizam-se as classes do JavaGroups que tratam com tarefas a um nível de abstração mais alto que o Canal e são usadas para auxiliar o programador da aplicação na construção de aplicações distribuídas tolerantes a falhas. Alguns dessas classes realizam tarefas como emular troca de mensagens síncronas sob sistemas assíncronos (Invocação Síncrona), utilização de filas para troca de dados entre *threads*, ou chamada a um objeto remoto específico (Chamada a Método Remoto), entre outras.

Abaixo do Canal encontram-se os protocolos utilizados para o transporte confiável de mensagens. Esses protocolos são estruturados em camadas, como no sistema iBus, e realizam tarefas como ordenação *fifo* (camada FIFO), ordenação total (camada TOTAL) e comunicação *multicast* (camada UDP), entre outros. Além da pilha de camadas de protocolos, o JavaGroups possui interface para os sistemas Ensemble (*Interface* Ensemble) e iBus (*Interface* iBus). Dessa forma, um objeto aplicação pode se beneficiar das classes intermediárias do JavaGroups e utilizar por exemplo, o iBus como camada de transporte confiável. O JavaGroups apresenta a vantagem de também ser escrito totalmente em Java.

5.5 – Conclusão

Neste capítulo apresentamos outras abordagens para adicionar Tolerância a Falhas às aplicações Java utilizando recursos da linguagem. Ambientes como o *Filterfresh*, o ROAPI e o JavaGroups dão suporte à construção de aplicações Java tolerantes a falhas utilizando diferentes abordagens. O *Filterfresh* é um ambiente que adiciona Tolerância a Falhas ao RMI do Java através da replicação dos objetos servidores e do Registrador. O ROAPI é uma API para a construção de objetos Java replicados e que pode ser utilizada sobre os ambientes RMI e iBus.

O JavaGroups é um Sistema de Comunicação em Grupo escrito totalmente em Java e estruturado em camadas. A principal característica do JavaGroups é a presença de classes Java que facilitam o trabalho do programador da aplicação. O JavaGroups possui interface para os sistemas Ensemble e iBus. Esses sistemas podem ser utilizados como camada de transporte para o envio e recebimento confiável de mensagens

Comparando algumas características como nível de abstração e desempenho do Filterfresh, ROAPI e do JavaGroups em relação ao iBusTF, obtivemos as seguintes conclusões. Em relação a implementação, os ambientes implementados possuem quase todas as principais propriedades para introduzir Tolerância a Falhas aos objetos Java através de comunicação

em grupo, como comunicação *multicast*, ordenação total e um protocolo de *membership* para manter a visão dos membros dos grupos consistentes.

No entanto, o iBusTF é o único sistema que possui um mecanismo de acordo sobre as suspeitas de falhas. Apenas o iBusTF realiza um acordo entre os membros operacionais do grupo, antes de remover um membro suspeito de falha. No JavaGroups a mudança de Visão do grupo é gerenciada pelo coordenador do grupo. Este se encarrega de enviar uma mensagem ao grupo e receber um confirmação de todos os membros sobre a mudança na configuração do grupo. Este procedimento pode comprometer o desempenho, pois se o coordenador falhar durante este procedimento, haverá um certo atraso até a escolha do novo coordenador e prosseguimento da mudança de visão do grupo. O JavaGroups encontra-se em fase inicial de implementação e muitas tarefas encontram-se apenas especificadas. Após a implementação de todas propriedades que encontram-se especificadas, este poderá apresentar mais funcionalidades que o iBusTF.

Comparando o Filterfresh em relação ao iBusTF, este apresenta uma restrição em relação ao iBusTF: o Filterfresh não oferece suporte à replicação de objetos clientes, apenas objetos servidores. O modelo de comunicação no Filterfresh é do tipo *Request/Reply* onde um cliente invoca uma chamada no servidor, enquanto no iBusTF o modelo de comunicação é do tipo *Publish/Subscribe*, onde uma mensagem é enviada a todos os objetos inscritos no grupo. Enquanto no Filterfresh cliente e servidor têm funções bem definidas, no iBusTF os mesmos objetos podem ser clientes e servidores e dessa maneira qualquer objeto pode ser replicado. Outra desvantagem do Filterfresh é que a comunicação *multicast* entre os servidores replicados é implementada através de vários *unicast*, o que pode levar uma sobrecarga nos canais de comunicação. A vantagem do Filterfresh em relação aos outros sistemas é que este apresenta um nível de abstração melhor, ou seja, a comunicação entre o cliente e o servidor é quase transparente para o cliente.

Outro ponto importante a ser observado, é que ROAPI e o JavaGroups possuem interface para o iBus, ou seja, o iBus pode ser utilizado em conjunto com o JavaGroups e com o

ROAPI. Assim como o iBus, o JavaGroups e o ROAPI também podem ser utilizados em conjunto com o iBusTF, visto que as alterações realizadas no iBus para adição dos protocolos de Ordenação Total e Reconfiguração de Grupos não comprometem sua interação com os ambientes que utilizam o iBus. O iBusTF permite aos sistemas que utilizam o iBus, obterem mais funcionalidades sem a necessidade de nenhuma alteração em sua interface.

Como o Filterfresh, o ROAPI e o JavaGroups encontram-se em fase inicial de implementação, não houve a possibilidade, durante o desenvolvimento desta dissertação, de comparar o desempenho destes quatro ambientes.

Essa dissertação apresentou uma abordagem para Tolerância a Falhas em Java através de Comunicação em Grupo. O desenvolvimento deste trabalho envolveu o estudo de diversos tópicos relativos a Sistemas Distribuídos, Tolerância a Falhas, Comunicação em Grupo e da linguagem Java, no intuito de definir a solução mais adequada para a construção de aplicações Java Tolerantes a Falhas.

O ambiente desenvolvido, denominado iBusTF (iBus Tolerante a Falhas), é uma extensão ao sistema iBus, desenvolvido por Silvano Maffeis[MAF96]. O iBusTF adicionou ao iBus as propriedades da Comunicação em Grupo fundamentais para manter a consistência num grupo de réplicas ativas: ordenação total e *membership atômico*. As alterações realizadas no iBus mantiveram total compatibilidade com o iBus, não necessitando alterações na interface da aplicação iBus para utilizar o iBusTF.

Além do suporte a replicação ativa de componentes, outros requisitos foram estabelecidos por nós para introduzir Tolerância a Falhas a Java: preservar a portabilidade e manter a flexibilidade para inserção de novas funcionalidades. Nesse sentido, discutimos as vantagens do iBusTF em relação a outras abordagens por nós analisadas (alterando a classe Java *MulticastSocket*, estendendo o RMI com suporte a grupos e utilizando um serviço de comunicação em grupo disponível numa plataforma CORBA).

A classe Java *MulticastSocket* suporta comunicação *multicast* mas sem garantias de ordenação ou reenvio de mensagens perdidas. Discutimos e propomos alternativas para estender a classe *MulticastSocket* com as funcionalidades básicas da Comunicação em Grupo, tanto adicionando essas funcionalidades diretamente ao código da classe *MulticastSocket*, como através de uma nova classe denominada *ReliableMulticastSocket*, localizada acima da classe *MulticastSocket*.

No RMI propomos a inserção de uma nova camada na arquitetura do RMI, denominada gerenciador de grupos, onde seriam implementadas as principais propriedades da Comunicação em Grupo.

A última abordagem discutida utiliza um Serviço de Comunicação em Grupo de uma plataforma CORBA para adicionar Tolerância a falhas às aplicações Java. Alguns ambientes de Comunicação em Grupo em CORBA foram apresentados na seção 2.3.4.2.

Não adotamos as abordagens de estender o RMI com suporte a grupos e implementar uma nova camada denominada *ReliableMulticastSocket*, em face de algumas desvantagens encontradas nas mesmas. Na abordagem para tolerância a falhas em Java através do RMI a comunicação *multicast* é implementada através de vários *unicasts*, o que pode levar a uma sobrecarga nos canais de comunicação. Na outra abordagem, baseada na classe *ReliableMulticastSocket*, se for preciso adicionar novas funcionalidades a esta classe, precisaríamos modificar todo o seu código.

Ao invés de construirmos então uma classe única, nos concentramos nesse trabalho, na implementação de um conjunto de classes Java que suporta as propriedades da Comunicação em Grupo, como ordenação total, e *membership atômico* e que se beneficia da classe Java *MulticastSocket* para comunicação *multicast*. O processo de definição desse ambiente em Java envolveu a análise de diversas propostas referentes à Tolerância a Falhas: ISIS[BCJ+91], Horus[RMB96], Newtop[MES93, MAC94, EMS95], Orbix + ISIS[IONA95], e Electra[MAFa95]. Um resumo desse estudo foi apresentado no capítulo 2.

Após as diversas pesquisas realizadas, encontramos um ambiente que contemplava alguns dos requisitos por nós exigidos para um ambiente Java tolerante a falhas. Esse ambiente, denominado iBus[MAF96, MEMa99, MEMb99, SW99], é implementado totalmente em Java, oferece suporte a comunicação *multicast* utilizando a classe Java *MulticastSocket*, e é estruturado em camadas permitindo a adição de novos protocolos. Além dessas características, o iBus oferece suporte básico a grupos, ordenação FIFO, protocolos para fragmentação de mensagens, reenvio de mensagens perdidas e um mecanismo de detecção de falhas baseado em *timeouts*.

O iBusTF implementado nessa dissertação adicionou uma nova camada ao sistema iBus[MAF96] onde foram implementados os protocolos de ordenação total e

membership atômico baseados no modelo da BCG [MES93, MAC94]. Esses protocolos são importantes para manter as visões dos membros dos grupos consistentes em modelos de replicação ativa

O protocolo de ordenação total assegura que as mensagens serão entregues aos membros dos grupos na mesma ordem global. O protocolo *membership* atômico assegura a atomicidade nos eventos de mudança de visão (inserção e remoção de objetos nos grupos). O protocolo de *memberhip* atômico implementado nessa Dissertação estabelece que um membro suspeito de falha será retirado do grupo apenas após um acordo entre os membros não suspeitos. Esse procedimento assegura que todos os membros operacionais de um grupo permanecerão com a mesma visão do grupo mesmo após a saída e a falha de membros.

Depois de confirmada a importância de inserir essas novas funcionalidades no iBus, passamos a observar como inseri-las sem contudo, comprometer o funcionamento do iBus, de maneira que para os clientes iBus a presença dessas duas novas propriedades fosse transparente. Os protocolos de Ordenação Total e *membership* atômico adicionados ao iBus são baseados nos protocolos da BCG (Base Confiável de Comunicação em Grupo), em desenvolvimento no Laboratório de Sistemas Distribuídos (LaSiD). A BCG[GM98] é uma plataforma de Comunicação em Grupo com suporte ao desenvolvimento de aplicações distribuídas confiáveis.

O próximo desafio a ser transposto foi a inserção de novas funcionalidades ao iBus de forma transparente para os clientes do iBus. Isso foi alcançado e permitiu aos usuários do iBus utilizar esse novo ambiente, denominado iBusTF (iBus Tolerante a Falhas), da mesma forma que utilizavam o iBus. Para os clientes iBus se beneficiarem da camada TF, basta especificar esta camada quando da criação da pilha.

As principais alterações realizadas no iBus para que obtivéssemos o iBusTF foram: a inclusão de uma nova camada denominada TF, a adição de um identificador único para cada objeto do grupo e a criação de novos eventos. A camada TF, que implementa os protocolos de ordenação total e *membership* atômico, foi introduzida abaixo da camada

STACK do iBus e acima da camada FRAG. O identificador (ID) único foi utilizado com o propósito de diferenciar os membros dos grupos nas estruturas de dados (vetores e matriz) auxiliares na ordenação de mensagens e na manutenção das visões.

Novos eventos também foram criados, como o evento nulo (*ev.MessageNull*), que são mensagens nulas com a função de completar blocos na Matriz de blocos e os eventos utilizados para realização do acordo sobre as falhas de membros. O evento de suspeita (*ev.suspect*), é utilizado para comunicar ao grupo sobre a suspeita de falha de algum membro do grupo. O evento de refute (*ev.refute*) refuta um evento de suspeita enviado. O evento de recuperação de mensagens (*ev.recovery*) é utilizado para solicitar o envio de mensagens não recebidas e o evento de confirmação (*ev.confirmed*) é utilizado para confirmar a suspeita de falha de um membro.

Realizamos alguns testes de desempenho no iBusTF em quatro PC`s 300Mhz, conectados por uma rede *Ethernet* 10Mbps, sistema operacional Windows 95 e JDK1.2.2. O *Round Trip* no iBusTF obteve um *overhead* maior que o iBus, visto que para garantir a ordenação total há um atraso no envio de mensagens à aplicação. Num segundo teste, forçamos a parada - crash - de um membro do grupo. O tempo obtido nesse teste foi a diferença entre a suspeita de falha de um membro até a sua completa remoção do grupo. Esse tempo também aumenta o *overhead* no iBusTF, visto que durante os eventos de mudança de visão, a *thread Delivery* bloqueia o envio de mensagens à aplicação. Esses *overheads*, no entanto, são necessários para atender os fortes requisitos de consistência de replicação ativa.

Os resultados obtidos com o ambiente iBusTF contemplam nossa proposta inicial no que tangia ao desenvolvimento desse trabalho: introduzir tolerância a falhas às aplicações JAVA utilizando a técnica de replicação ativa e preservando a portabilidade da linguagem. Adicionalmente, esse trabalho também contribuiu com a exposição de outras possíveis abordagens para introduzir Tolerância a Falhas à Java, e com a apresentação de outros trabalhos em desenvolvimento nessa área.

6.1 – Trabalhos Futuros

O prosseguimento desse trabalho pode ser realizado visando a introdução de novos membros nos grupos após a formação inicial da Matriz de Blocos [MAC94]. Além de suportar a inserção de novos membros, também é fundamental a presença de um protocolo de Transferência de Estado para que os novos membros recebam as mensagens que foram enviadas anteriormente à sua entrada. Essas alterações são muito importantes para que o iBusTF possa ser disponibilizado aos usuários iBus via Internet.

Outra continuação desse trabalho pode ser a adição de uma nova camada para implementar um protocolo de ordenação causal de mensagens[MACb95, LMb99], que possui um custo menor que a ordenação total.

Atualmente o iBus está sendo comercializado e por isso os desenvolvedores estão em contínuo aprimoramento de suas funcionalidades. Algumas dessas novas atualizações nas versões seguintes ao iBus 0.3 foram: interface para objetos escritos em C e C++, interface para Java Beans e suporte aos protocolos de comunicação TCP/IP para comunicação entre *intranets*, e *wireless* para comunicação via satélite.

Referências Bibliográficas

[ADM+92] Yair Amir, Danny Dolev, Shlomo Kramer and Dalia Malki. *Transis: a communication sub-system for high availability*. In proceedings of the 22nd Annual International Sysposium on Fault-Tolerant Computing. Boston, July 1992.

[BAN98] Bela Ban. *Design and Implementation of a Reliable Group Communication Toolkit for Java*. Dept. of Computer Science. Cornell University. December 1998. http://www.cs.cornell.edu/home/bba/papers.html

[BAN99] Bela Ban. *JavaGroups User's Guide*. Dept. of Computer Science. Cornell University. March 1999.

[BCJ+90] Kenneth P. Birman, R. Cooper, T A Joseph, K Marzullo, M Makpangou, K kane, F Schumuk, and M Wood. *The Isis System Manual, Version 2.0*. Dept of Computer Science, Cornell University, March 1990.

[BHM99] N. Badache, M. Huffin and R. Macêdo. *Solving the Consensus Problem in a Mobile Environment*. The 1999 International Performance, Computing and Communications Conference – IPCCC'99, Phoenix, USA, Feb/99. IEEE Computer Society.

[BIR87] K. Birman and T. Joseph. *Reliable Communication in the Presence of Failures*. ACM Transactions on Computer Systems, 5 (1):47-76 February 1997.

[BIR91] K. Birman, A. Schipper, P Stephenson. *Lightweight Causal and Atomic Group Multicast*. ACM Transactions On Computer Systems, Vol 9. N° 3 August 1991, pp.272-314.

[BIR93] Kenneth P Birman. *The Process Group Approach to Reliable Distributed Computing*. Communications of ACM, 36(12):37-53. December 1993.

[BIR94] Kenneth P Birman. *Virtual Synchronous Model*. Reliable Distributed Computing with the ISIS Toolkit. IEEE Computer Society, 1994.

[BIR96] Kenneth P Birman. *Building Secure and Reliable Network Applications*. Manning Publications CO., 1996.

[BIR99] Kenneth P Birman. *New Prospects for Secure and Reliable Network Technologies*. 17° Simpósio Brasileiro de Redes de Computadores. SBRC99. Salvador. Maio, 1999.

[BJ87] K. Birman and T. Joseph. *Exploiting Virtual Synchrony in Distributed Systems*. The proceedings of the 15th ACM Symposium on Operating Systems Principles. Pages 123-138, Ausyin Texas, November 1987.

[BR94] Kenneth P Birman Robert Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press. ISBN 0-8186-5342-6.

[CDK94] George Coulouris, Jean Dollimore, Tim Kindberg. *Distributed Systems – Concepts and Design*. Second Edition. Addison-Wesley, 1994.

[CH97] Gary Cornell, Cay S. Horstmann. *Core Java*. Makron Books, 1997.

[CHR+99] P. Emerald Chung, Yennun Huang, Sampath Rangarajan and Shalini Yajnik *Filterfresh: Hot Replication of Java RMI Server Objects*. Bell Laboratories. Lucent Technologies, USA.

[CHT95] T. Chandra, V. Hadzilacos and S. Toueg. *Impossibility of group membership in asynchronous systems*. Technical Report 95-1533, Computer Science Department, Cornell University, August 1995.

[ECK98] Bruce Eckel. *Thinking in Java*. ISBN 0-13-659723-8 Prentice Hall PTR 1998

[EJB99] Sun Microsystems Inc. *Enterprise JavaBeans Specification 1.1*. http://java.sun.com/products/ejb/newspec.html

[EMS95] Paul Ezhilchelvan, Raimundo A Macêdo, Santosh K Shrivastava. *Newtop: a Faul-Tolerant Group Communication Protocol*. Proceedings of the 15th International Conference on Distributed Computing Systems. IEEE Computer Society. Pages 296-306, Vancouver - Canada. May 30-June 2, 1995.

[FG96] Pascal Felber and Rachid Guerraoui. *Programming with Object Groups in Phoenix*. Technical Report CH-1015. Lausanne, Switzerland

[FGG96] P. Felber, B. Garbinato and R. Guerraoui. *The Design of a CORBA Group Communication Service*. 15th Symposium on Reliable Distributed Systems, pp 150-159, October 1996.

[FR95] R. Friedman and Robert Van Renesse. *Strong and Weak Virtual Synchrony in Horus*. Technical Report TR 95-1491, Department of Computer Science, Cornell University. March 1995.

[GM98] Fabíola Greve, Raimundo A Macêdo. *The BCG Membership Service Performance Analysis*. Anais do XVI Simpósio Brasileiro de Redes de Computadores – SBRC98. Pp. 682-700. Rio de Janeiro, Maio 1998.

[MAC98] R. J. Macêdo. *Comunicação em Grupo e Sincronismo Virtual: Aspectos da Plataforma BCG*. I Open Workshop of the Logic for Concurrency and Synchronism–LOCUS–Project,UFPE.Março/1998. http://www.di.ufpe.br/~locus.

[MAFa95] Silvano Maffeis. *Adding Group Communication and Fault-Tolerance to CORBA*, In Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies. Monterey, VA. June 1995.

[MAFb95] Silvano Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. Ph.D. Thesis University of Zurich. Zurich: 1995.

[MAF96] Silvano Maffeis. *iBus – Java Intranet Software Bus* (http://www.softwired.ch/people/maffeis).

[MEMa99] Altherr Marcel, Martin Erzrberger and Silvano Maffeis. *iBus – A Software Bus for the Java Plataform*. Java Report. September, 1999.

[MEMb99] Altherr Marcel, Martin Erzrberger and Silvano Maffeis. *Electronic Business– A Case for Messaging Middleware?*. Java Developer Journal. June, 1999.

[MES93] R. J. Macêdo P. Ezhilchelvan and S. Shrivastava. *Flow Control Schemes for Fault Tolerant Multicast Protocols*. The proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS'95), December 4-5, 1995. Newport Beach, California, USA. IEEE Computer Society.

[MES95] R. J. Macêdo P. Ezhilchelvan and S. Shrivastava. *Newtop: a Total Order Multicast Protocol Using Causal Blocks*. First Year Report – Fundamental Concepts 1 of 3, BROADCAST ESPRIT Basic Research Project 6360, 1993.

[MSa95] R. J. Macêdo and P. Ezhilchelvan. *The Implementation and Performance Analysis of a Total Order Delivery Protocol for Group Communication*. The proceedings of XXI Latin American Conference on Informatics and the XV Congress of the Brazilian Computer Society. Pages 287-299, July, 1995, Canela-RS, Brazil.

[MSb95] R. J. Macêdo and P. Ezhilchelvan. *Reliability Aspects of Multicast Protocols*. Anais do VI Simpósio Brasileiro de Computadores Tolerantes a Falhas. Pags. 239 a 259. Julho, 1995. Canela-RS.

[OH98] Robert Orfali, Dan Harkey. *ClientSever Programming with Java and CORBA*. Second Edition.1998

[OMG95] OMG. *The Commom Object Request Broker: Architecture and Specification*. OMG, 1995. Revision 2.0.

[RBF+ 95] R. V. Renesse, K. P. Birman, R. Friedman, M. Hayden and D Karr. *A Framework for Protocol Composition in Horus*. In p.roc. Of the 14th Symposium on Principles of Distributed Computing, August 1995.

[RBH94] R. V. Renesse, K. P. Birman and T. M. Hickey. *Design and Performance of Horus: A Lightweight Group Communications System*. Technical Report 94-1442, Cornell University. Dept. of Computer Science, August 1994.

[RMB96] Robert Van Renesse, Silvano Maffeis and Kenneth P. Birman. *Horus: A Flexible Group Communication System*. Communications of the ACM. April, 1996.

[RMI96] Sun Microsystems Inc. *Java Remote Method Invocation Specification*. 1.1 edition, November 1996. Draft.

[SCH90] Fred B. Schneider. *Replication management Using the State Machine Approach*. ACM Computing Surveys. Pg22. December 1990.

[SER99] Sun Microsystems Inc. *Java Servlet API Reference, v2.2*. http://java.sun.com/products/servlet/2.2/

[SUN99] Sun Microsystems Inc. *The Java Tutorial*. http://java.sun.com/docs/books/tutorial/index.html

[SW99] SoftWired Inc. (http://www.softwired.inc.com)

[VRV93] Paulo Veríssimo, Luís Rodrigues and Werner Vogels. *Group Orientation: a paradigm for Modern Distributed Systems*. BROADCAST Project deliverable report, vol I, October 1993.