Uma Abordagem para Tolerância a Falhas em JAVA através de Comunicação em Grupo

Lilianne Dantas Cirne[1], Raimundo José de Araújo Macêdo[2]

Universidade Federal da Bahia
Laboratório de Sistemas Distribuídos – LaSiD
Prédio do CPD, Av. Adhemar de Barros, S/N, 401170-110. Salvador-BA
[1]{lilianne@lasid.ufba.br} [2]{macedo@ufba.br}

Resumo

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. A técnica de replicação ativa de componentes é freqüentemente utilizada quando se quer implementar tais serviços de alta disponibilidade e tolerantes a falhas. Este artigo apresenta uma proposta para adição de Tolerância a Falhas ao ambiente JAVA capaz de suprir as necessidades da replicação ativa. Para atingir esse objetivo, desenvolvemos uma extensão ao sistema iBus, projetado por Silvano Maffeis[MAF96]. 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 m*embership atômico*. A abordagem adotada tem a vantagem de somente usar recursos já disponíveis em JAVA, mantendo total compatibilidade com o sistema iBus.

Abstract

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. Active replication is usually used when one aims at building such high available and fault-tolerant services. 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.

1. Introdução

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, não devem interromper seu funcionamento correto mesmo na presença de falhas de alguns componentes. As técnicas utilizadas para prover Tolerância a Falhas caracterizam-se pela redundância: a presença de componentes

---

[1] Aluna de Mestrado em Ciência da Computação. COPIN - Universidade Federal da Paraíba.
[2] Pesquisador do CNPq, número da bolsa 300013/87-6 (Projeto BCG).

(*hardware* e *software*) ou informações replicadas e estruturadas em grupos. 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 seção 2).

Alguns Sistemas de Comunicação em Grupo já oferecem suporte para o desenvolvimento de aplicações distribuídas tolerantes a falhas, como o ISIS [BIR87, BCJ^{+}90], Horus [RBH94] e BCG [MAC95, GM98], entre outros. Todavia, a maioria desses sistemas não são portáveis, uma característica muito importante em sistemas distribuídos, visto que a maioria das empresas utilizam ambientes de computação heterogêneos, compostos por produtos de *hardware* e *software* de diferentes fabricantes.

Nesse contexto, a linguagem JAVA vem se destacando devido principalmente à sua portabilidade e suporte para aplicações distribuídas. Para garantir a portabilidade, o compilador JAVA gera um código denominado JAVA *bytecodes* que pode ser executado em diferentes plataformas desde que estas possuam o interpretador JAVA, também denominado Máquina Virtual JAVA (JAVA *Virtual Machine*)[SUN99]. Além dessa característica, a linguagem JAVA é orientada a objetos e distribuída[CH97]. Ela oferece suporte para a comunicação entre os objetos remotos através dos protocolos TCP, UDP e do *Remote Method Invocation* (RMI)[RMI96]. Apesar de todas essas vantagens, a linguagem JAVA não oferece suporte a Tolerância a Falhas.

Este artigo apresenta uma proposta para adição de Tolerância a Falhas à JAVA capaz de suprir as necessidades de replicação ativa, vistas anteriormente. Para atingir esse objetivo, desenvolvemos uma extensão ao sistema iBus versão 0.3[3], projetado por Silvano Maffeis [MAF96, MEMa99, MEMb99]. O sistema desenvolvido, denominado iBusTF (iBus Tolerante a Falhas), possui as propriedades da Comunicação em Grupo fundamentais para manter a consistência num grupo de réplicas ativas: Ordenação Total e *Membership atômico* (propriedades essas ausentes no iBus versão 0.3). A abordagem adotada tem a vantagem de somente usar recursos já disponíveis em JAVA, mantendo total compatibilidade com o sistema iBus original.

Ao adicionarmos mecanismos de Tolerância a Falhas a JAVA, buscamos fazê-lo de forma a preservar sua portabilidade (não modificar código), mantendo a flexibilidade necessária para composição de novas funcionalidades. Nesse sentido, discutimos e contrapomos nossa abordagem com outras possibilidades tais como introduzir novas funcionalidades à classe JAVA *MulticastSocket*[SUN99], estender o *Remote Method*

[3] Primeira versão *freeware* e aberta, disponibilizada por Silvano Maffeis[MAF96], que permitiu a inserção de novos módulos a sua estrtura.

Invocation (RMI) do JAVA com suporte a grupos ou utilizar um serviço de Comunicação em Grupo disponível em outros ambientes, como na plataforma ORBIX+ISIS[IONA95] e Electra[MAF96]. A escolha pelo iBus deu-se pelo fato deste já oferecer algumas características básicas por nós desejadas para um sistema de Comunicação em Grupo (ordenação *fifo*, suspeita de falhas baseado em *timeouts* e comunicação *multicast*). Além disso, ele possui uma arquitetura flexível, em camadas, que nos permitiu estender suas funcionalidades sem comprometer a flexibilidade e portabilidade almejadas.

Para atender aos requisitos da replicação ativa, incluímos ao iBus novos módulos responsáveis pela ordenação de mensagens e *membership atômico.* Os protocolos de ordenação total e *membership atômico* utilizados são baseados no modelo proposto no sistema BCG (Base Confiável de Comunicação em Grupo) desenvolvido no LaSiD/UFBA (Laboratório de Sistemas Distribuídos). A descrição dos fundamentos e algoritmos relacionados com a BCG estão disponíveis em outras publicações [MAC98, GM98, MES93, MES95]. Esse artigo não se preocupa, portanto, em mostrar tais fundamentos e algoritmos, mas em descrever sua utilização num ambiente JAVA. Como visto anteriormente, esses protocolos foram implementados em uma nova camada denominada TF e adicionados à arquitetura do iBus sem, contudo, modificar a sua estrutura principal, permitindo que os usuários do iBus utilizem o iBusTF sem a necessidade de alterações na estrutura de seus programas.

O texto encontra-se estruturado da seguinte forma: a seção 2, descreve as propriedades básicas necessárias para prover Comunicação em Grupo. A seção 3, discute algumas abordagens para adicionar Tolerância a Falhas a JAVA e descreve a solução adotada: o ambiente iBusTF. Nesta seção também são apresentados detalhes da implementação a alguns dados de desempenho. A seção 4 discute trabalhos correlatos e a seção 5 apresenta as conclusões e contribuições obtidas neste trabalho.

2. Tolerância a Falhas em Sistemas Distribuídos através de Comunicação em Grupo

O objetivo da Tolerância a Falhas é garantir a continuidade do serviço oferecido mesmo na presença de falhas de alguns componentes. As técnicas para introduzir tolerância a falhas são caracterizadas pela redundância, onde os componentes (*hardware* e *software*) ou informações são replicadas e estruturados em grupos. Os grupos constituem-se uma maneira conveniente de endereçar os componentes redundantes, sem a necessidade da localização explícita de cada componente. 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: Endereço de Grupo, Atomicidade, Ordenação, *Membership* e Sincronismo Virtual [BIR93].

O endereço de grupo pode ser definido como contendo os endereços dos membros do grupo (tabela de endereços), permitindo um objeto enviar mensagens a um grupo sem o conhecimento prévio dos seus membros (o número de membros e a sua localização). A propriedade de atomicidade garante que uma mensagem enviada a um grupo ou será recebida por todos os membros operacionais desse grupo, ou não será recebida por nenhum membro.

A ordenação de mensagens é necessária para se obter o comportamento correto dos membros dos grupos, sincronizando a ordem em que as ações são executadas. Na Ordenação Causal, apenas as mensagens não concorrentes são ordenadas. Na visão de Lamport[LAM78], dois eventos são concorrentes se eles não são relacionados pela relação de causa-efeito (*happened-before*). Na Ordenação Total, todos os membros dos grupos recebem as mesmas mensagens na mesma ordem. Nesse artigo, adotaremos a definição de Ordenação Total proposta em [MAC94], isto é, todos os objetos de um mesmo grupo entregam o mesmo conjunto de mensagens na mesma ordem, mantendo a relação causal entre as mensagens.

A propriedade de *membership* corresponde ao gerenciamento dos membros de um grupo: criação e destruição de grupos, inserção e remoção de membros. O *membership* também é responsável por manter atualizada a informação referente aos membros que compõem um grupo, informação esta denominada Visão do Grupo (*Group View*). No entanto, em sistemas assíncronos os tempos de transmissão das mensagens diferem de acordo com a carga do sistema, o que dificulta a resolução de problemas como sincronização, consistência e detecção de falhas em ambientes distribuídos. Para garantir então a consistência das mudanças de visões entre os membros dos grupos em sistemas assíncronos, foi introduzida a propriedade de Sincronismo Virtual[BIR94]. A propriedade de Sincronismo Virtual garante que todos os membros operacionais em um grupo observem os eventos de mudança de visão do grupo na mesma ordem. Nesse artigo adotaremos o termo *membership atômico* para denominar um protocolo de *membership* que atende as propriedades de Sincronismo Virtual descritas em [EMS95].

Os sistemas e plataformas que tratam com Comunicação em Grupo incorporam a maioria das propriedades descritas anteriormente. Alguns desses sistemas são o ISIS[BIR87, BCJ+90], Horus[RBH94], Newtop[MES93, MAC94, EMS95], Orbix + ISIS[IONA95] e a plataforma BCG[MAC95, GM98]. Uma das desvantagens da maioria dos sistemas de Comunicação em Grupo é que eles não são portáveis. A portabilidade é fundamental para integrar ambientes distintos, oriundos de diferentes fabricantes, principalmente em Sistemas Distribuídos. Ela permite que objetos sejam executados em diferentes sistemas, sem necessidade de modificações. O uso da linguagem JAVA na Comunicação em Grupo é visto como uma esperança para o problema da portabilidade.

3. Tolerância a Falhas em JAVA através de Comunicação em Grupo

A linguagem Java tem sido cada vez mais utilizada em Sistemas Distribuídos devido principalmente a sua independência de plataforma, que é fundamental para integrar ambientes distintos, oriundos de diferentes fabricantes principalmente com o advento da Internet. Além de portável, os projetistas da linguagem Java a descrevem como uma linguagem simples, neutra em relação à arquitetura e *multithreading*.

3.1. Comunicação entre Objetos Remotos em Java

A comunicação entre objetos remotos em JAVA pode ser tanto via *Socket,* utilizando os protocolos de comunicação TCP e UDP, como em um nível superior de abstração através do *Remote Method Invocation*(RMI)[RMI96]. O protocolo TCP é orientado a conexão e provê suporte para transmissão confiável de mensagens entre pares de objetos, um objeto cliente e um objeto servidor. Apesar do TCP oferecer certas garantias como ordenação *fifo* e reenvio de mensagens perdidas, este não oferece suporte a *multicast*. Por outro lado, através do protocolo UDP as mensagens são enviadas de uma máquina a outra sem garantias de que essas mensagens irão atingir o destino (as mensagens são apenas enviadas ao endereço especificado, sem uma conexão preestabelecida). Porém, utilizando o protocolo UDP, é possível se beneficiar da facilidade de *multicast*, ou seja, da comunicação de um para vários objetos. A classe *MulticastSocket* do JAVA permite que objetos se juntem a um grupo e recebam os *datagramas* enviados ao endereço de *multicast*.

O RMI é baseado no modelo de comunicação requisição-resposta do RPC (*Remote Procedure Call*), onde uma chamada a um objeto remoto possui a mesma sintaxe que uma chamada a um objeto local. Nos mecanismos de RPC, assim como no RMI, uma grande parte da tarefa de comunicação é feita por rotinas especiais conhecidas por *stubs* presentes no cliente e no servidor. No RMI o *stub* do servidor é denominado *skeleton*. O RMI dispõe também de um serviço de nomes ou Registrador, que permite a localização dos objetos servidores remotos. Este serviço de nomes permite aos clientes obterem as referências dos objetos servidores. A arquitetura do RMI é estruturada em três camadas: a Camada S*tub/Skeleton*, a Camada de Referência Remota ou RRL (*Remote Reference Layer*) e a Camada de Transporte (*Transport Layer*), como mostra a figura 01.

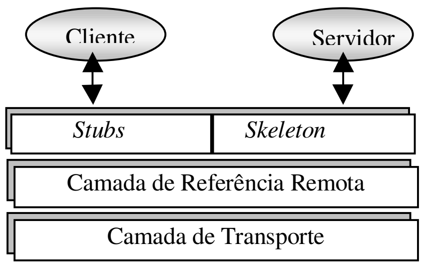

Figura 01: Arquitetura do RMI

A camada *Stub/Skeleton* é a interface entre a camada de aplicação e as outras camadas do sistema RMI. As informações são enviadas do s*tub* para a Camada de Referência Remota e desta para a Camada de Transporte, sendo então enviadas pela rede. Do lado servidor as informações trafegam entre as camadas até atingir o *skeleton*. O *skeleton* extrai as informação contidas no pacote enviado pelo cliente e entrega essas informação ao objeto servidor que vai executar o método remoto.

Caso ocorra alguma falha durante o processamento do método no objeto remoto, o cliente receberá como resposta uma exceção ou erro, ou seja, o RMI não está preparado para suportar falhas dos objetos servidores. Apesar das vantagens da linguagem Java, esta não oferece suporte a Tolerância a Falhas. A seguir apresentamos algumas possibilidades para introduzir Tolerância a Falhas a JAVA.

3.2. Incorporando Tolerância a Falhas a JAVA

Nossa proposta neste trabalho é introduzir um mecanismo de tolerância a falhas a JAVA capaz de suprir os requisitos de replicação ativa (*active replication*), onde 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*). Além dessas propriedades, buscamos adicionar Tolerância a Falhas a JAVA preservando a portabilidade e mantendo a flexibilidade necessária para composição de novas funcionalidades. Para atingir estes objetivos, desenvolvemos o ambiente iBusTF (iBus Tolerante a Falhas), uma extensão ao sistema iBus projetado por Silvano Maffeis [MAF96, MEMa99, MEMb99]. O iBusTF (iBus Tolerante a Falhas) acrescentou ao iBus as propriedades de Comunicação em Grupo fundamentais para manter a consistência num grupo de réplicas ativas: *ordenação total* e *Membership atômico* (propriedades essas ausentes no iBus versão 0.3).

Outras possibilidades para introduzir tolerância a Falhas a JAVA também foram consideradas por nós, a saber: acrescentar novas funcionalidades à classe JAVA *MulticastSocket*[SUN99], estender o *Remote Method Invocation* (RMI) do JAVA com suporte a grupos e, finalmente, utilizar um serviço de Comunicação em Grupo disponível em outros ambientes - como na plataforma ORBIX+ISIS[IONA95] e Electra[MAF96] -. No entanto, essas abordagens não atendem, ao mesmo tempo, a todos os requisitos estabelecidos (portabilidade, flexibilidade para incorporar novas funcionalidades e suporte a replicação ativa). A seguir discutimos essas três alternativas e apontamos as desvantagens em relação ao iBusTF.

Introduzindo Tolerância a Falhas a Java através da classe *MulticastSocket*, adicionaríamos a essa classe novas funcionalidades, como atomicidade, ordenação, *membership* e sincronismo virtual, de maneira a tornar a comunicação entre objetos JAVA através da classe *MulticastSocket* confiável. No entanto, nesta abordagem, não há uma flexibilidade para inserção de novas funcionalidades, além de comprometer a portabilidade – já que passariam a existir 2 tipos de classes *MulticastSocket.* Até onde pôde ser verificado, não há trabalhos na literatura que contemplam esta possibilidade.

Ainda utilizando recursos disponíveis da linguagem JAVA, outra possibilidade para introduzir Tolerância a Falhas a JAVA é estender o RMI (*Remote Method Invocation*) com suporte a grupos de objetos. Para introduzir Tolerância a Falhas nos objetos servidores e no Registrador, é necessário replicá-los e possibilitar a comunicação entre as réplicas. A alteração no RMI para adicionar Tolerância a Falhas poderia ser feita através da introdução de uma nova camada que executasse as funções de gerenciar os grupos (criação de grupos, inserção e remoção de membros, envio das mensagens aos membros do grupo e ordenação das mensagens), ou seja, uma nova camada que suportasse a criação e a comunicação entre os objetos servidores replicados. Essa abordagem foi considerada no trabalho intitulado *Filterfresh*[CHR+99], com a diferença de que o Filterfresh alterou a camada RRL do RMI para o suporte a grupos. O Filterfresh será abordado em mais detalhes na seção 4.

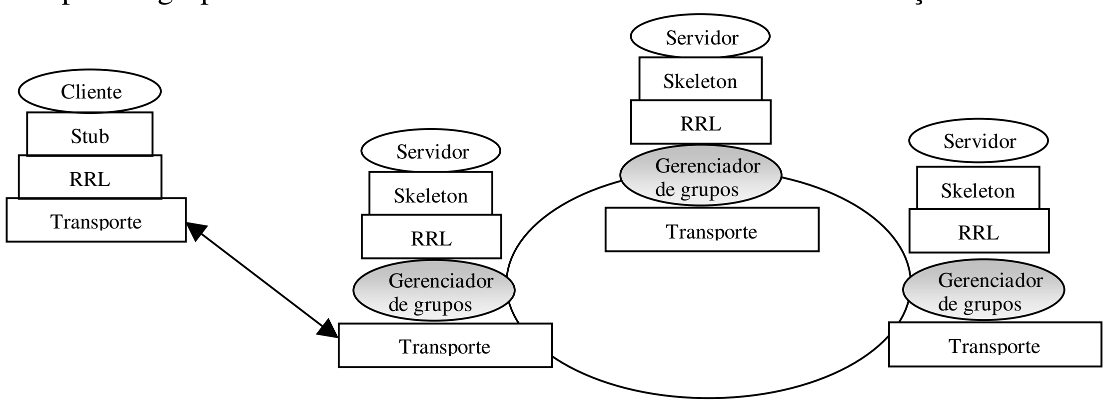

Figura 02: Replicação de objetos servidores no RMI

No exemplo da figura 02, adicionamos uma nova camada ao RMI denominada Gerenciador de Grupos, onde encontram-se as operações necessárias para criar e coordenar os objetos servidores. Esse novo RMI Tolerante a Falhas permite a criação e comunicação de objetos servidores replicados. Contudo, sua estrutura é modificada, podendo comprometer a portabilidade da linguagem. Outra desvantagem é, caso a estrutura do RMI seja alterada pela SUN em versões posteriores, o RMI Tolerante a Falhas também deveria ser reestruturado.

As abordagens apresentadas acima representam algumas vertentes para adicionar tolerância a falhas à JAVA num ambiente totalmente JAVA. Por outro lado, uma abordagem para construir aplicações JAVA Tolerantes a Falhas, e que não segue a linha das abordagens anteriormente descritas, é utilizar um serviço de Comunicação em Grupo disponível em outros ambientes. Sob este prisma, a plataforma CORBA seria a mais recomendável dada suas características de interoperabilidade.

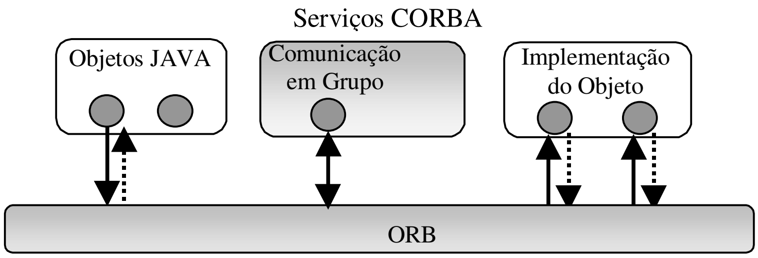

Figura 03: Objetos JAVA utilizando um Serviço de Comunicação em Grupo da plataforma CORBA.

A figura 03 apresenta objetos JAVA utilizando o serviço de Comunicação em Grupo de uma plataforma CORBA. Os sistemas Electra[MAFb95] e o Orbix + ISIS[IONA95] são exemplos onde o suporte a Comunicação em Grupo é integrado ao núcleo do ORB(*Object Request Broker*)[OMG95]. Em [FGG96], é proposto um Serviço de Comunicação em Grupo externo ao ORB, como demonstrado figura 03, onde a estrutura da plataforma CORBA não é alterada. Esta abordagem preserva a especificação da OMG (*Object Managment Group*)[OMG95] de garantir a interoperabilidade entre diferentes implementações de ORB's. Obviamente, adotar uma solução JAVA/CORBA foge ao nosso objetivo inicial de introduzir Tolerância a Falhas a um ambiente puramente JAVA.

A solução adotada foi acrescentar novas funcionalidades a um ambiente com algumas facilidades de comunicação em grupo e baseado puramente em JAVA (i.e., portável para qualquer ambiente JAVA). A escolha pelo iBus deu-se pelo fato deste já oferecer algumas características básicas importantes por nós almejadas. Ou seja, o iBus possui algumas propriedades da Comunicação em Grupo (ordenação *fifo*, suspeita de falhas baseado em *timeouts* e comunicação *multicast*) e possui uma arquitetura flexível, em camadas, que nos permitiu estender suas funcionalidades sem comprometer a flexibilidade e portabilidade. Denominamos o sistema resultante de iBusTF (iBus Tolerante a Falhas).

3.3. iBusTF - Uma abordagem para Tolerância a Falhas em Java

O iBusTF é formado por um conjunto de classes JAVA (*package* iBus) que pode trafegar pela rede e ser executado em qualquer plataforma que possua o interpretador JAVA. O iBusTF suporta comunicação assíncrona entre dois objetos ou entre um conjunto de objetos. O envio e recebimento de mensagens entre os objetos de um grupo é feita através de canais, e utiliza o paradigma *Publish/Subscribe*. Um objeto transmissor envia mensagens (*Publish*) a um canal e os objetos receptores inscritos nesse canal (*Subscribe*) recebem as mensagens.

A arquitetura do iBusTF permite à aplicação montar a sua pilha de protocolos de forma bastante flexível, a depender de suas necessidades (figura 04). Cada protocolo da pilha é responsável por uma função específica, como ordenação total e *membership atômico* (camada TF) e envio de dados (camada IPMCAST). A camada STACK, localizada no topo da pilha, é a interface entre a aplicação e o iBusTF.

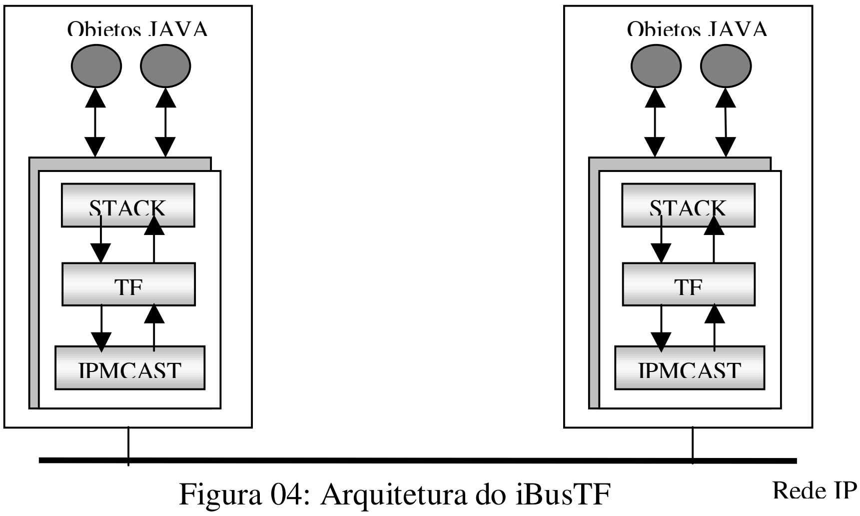

Figura 04: Arquitetura do iBusTF

A figura 04 mostra aplicações JAVA se comunicando através da pilha de camadas do iBusTF. As mensagens que trafegam ao longo das camadas são denominadas "eventos". Os eventos são utilizados para diferenciar mensagens envidas pela aplicação de mensagens envidas pelas camadas. As principais camadas do iBusTF são:

- **IPMCAST:** A camada IPMCAST é responsável pela transmissão de dados através dos protocolos de comunicação IP *multicast*, para envio a um grupo e UDP, para comunicação ponto-a-ponto.
- **REACH:** A camada REACH *(Reachability Membership Layer)* implementa um protocolo simples para gerenciamento de grupos e detecção de falhas. O protocolo de gerenciamento de grupos mantém uma tabela dos membros pertencentes ao grupo e a cada entrada/saída ou falha de um membro, é criado um evento denominado Visão (*event.View*) indicando a ocorrência (entrada, saída ou falha de membro).

O serviço de detecção de falhas da camada REACH é baseado em *timeouts*. Cada objeto membro do grupo envia sinais periódicos (*event.Heartbeat*) para cada grupo que este membro está inscrito. Se um membro permanece por um tempo predefinido sem enviar nenhum sinal, a camada REACH de outro membro que suspeitou da falha gera um evento Visão excluindo esse membro do canal e envia essa Visão para a camada superior. É importante observar que esta Visão pode não ser verdadeira, visto que é baseada apenas em sinais periódicos e o membro pode não ter falhado, mas o canal de comunicação estar lento.

- **NAK:** A camada NAK *(Negative Acknowledgement)* é responsável pelo reenvio de mensagens perdidas.
- **FIFO:** A camada FIFO é responsável pela ordenação *fifo (fisrt-in-first-out)* e por eliminar mensagens duplicadas.
- **FRAG:** A camada FRAG é responsável pela fragmentação e reconstrução de mensagens que ultrapassam um tamanho predefinido. O tamanho padrão é de um datagrama UDP.
- **TF:** A camada TF é responsável pela ordenação total e pela propriedade de *membership atômico*. Estas duas propriedades são necessárias para atender ao requisitos de replicação ativa. A seguir são apresentados detalhes da implementação correspondentes à inserção dos protocolos de ordenação total e *membership atômico* ao iBus.

3.3.1. Detalhes de Implementação do iBusTF

A implementação e inserção da camada TF a arquitetura do iBus foi realizada de maneira a não comprometer a estrutura do iBus, permitindo que esta camada seja utilizada em versões posteriores do iBus. Para atender a esses requisitos, a camada TF possui as operações comuns a todas as camadas, e também as operações necessárias para implementar os protocolos de ordenação total e *membership atômico*. A hierarquia de classes do iBusTF é ilustrada na figura abaixo, seguindo o padrão UML.

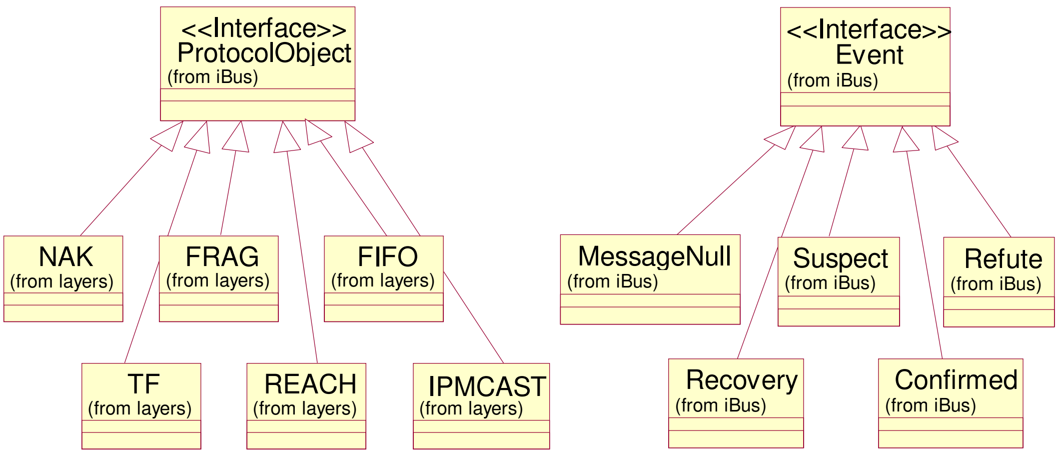

Figura 05: Hierarquia de classes do iBusTF

Todas as camadas do iBusTF herdam da interface *ProtocolObject*, e os eventos utilizados no iBusTF herdam da interface *Event*. Cada camada do iBusTF é um objeto Java e as mensagens são enviadas de uma camada a outra através de chamada a métodos. A classe principal da camada TF também é denominada TF. Nesta classe encontram-se as operações comuns a todas as camadas do iBus: *dnInit* (inicialização e empilhamento das camadas que serão utilizadas), *dnPush* (envio de eventos de cima para baixo da pilha), *upHandleEvent* (envio de eventos no sentido de baixo para cima na pilha) e as operações de *dnSubscribe* e *dnRegisterTalker* utilizadas quando um objeto deseja fazer parte de um grupo. Na classe TF encontram-se também as operações de *Agreement* (acordo sobre a suspeita de falha de membros) e *updateView* (atualização na composição do grupo).

Além da classe TF, foram criadas novas classes que tratam com os eventos utilizados pela camada TF: os eventos de *MessageNull* (mensagens nulas enviadas pela camada TF quando um membro permanece por um longo intervalo de tempo sem enviar mensagens), *Suspect* (suspeita de falha de membros), *Refute* (negação da suspeita de falha de membros), *Recovery* (recuperação de mensagens não recebidas) e *Confirmed* (confirmação da suspeita de falha de membros). Todos os eventos estão inseridos na operação *upHandeEvent*, e são identificados com um número único. Esses eventos, com exceção do *MessageNull*, são utilizados no protocolo de *membership atômico* e serão abordados novamente no decorrer desta seção.

O protocolo de Ordenação Total garante que todos os membros dos grupos irão receber as mesmas mensagens numa mesma ordem. Para atender a este requisito foi utilizada uma estrutura de dados denominada Matriz de Blocos[MES93][MAC94], onde são guardadas as informações referentes à ordenação das mensagens enviadas e recebidas no grupo. Cada objeto membro do grupo possui uma Matriz de Blocos local, que armazena as mensagens enviadas pela aplicação e recebidas no grupo. A entrega das mensagens à aplicação é implementada através de uma *thread* denominada *Delivery* (figura 06). As mensagens são removidas da Matriz de Blocos e entregues à aplicação seguindo uma ordem preestabelecida. A *thread Delivery* é "despertada" quando do recebimento de mensagens, verificando se as mensagens podem ser removidas da Matriz de Blocos e enviadas a aplicação. A *thread* LTS (*Local Time Silence*) é responsável por enviar mensagens eventos nulos (*evMessageNull*) com o objetivo de completar blocos[4].

Uma mensagem enviada pela aplicação passa primeiramente pela camada STACK, que é a interface entre a aplicação e o iBusTF, e desta para a camada TF através da operação *dnPush* (figura 06). Na camada TF essa mensagem é desempacotada e são adicionadas novas informações ao cabeçalho da mensagem. No caminho contrário, as mensagens são enviadas da camada inferior à camada TF através de uma chamada a operação *upHandleEvent*. As mensagens são então armazenadas na Matriz de Blocos e posteriormente entregues à aplicação seguindo uma ordem preestabelecida.

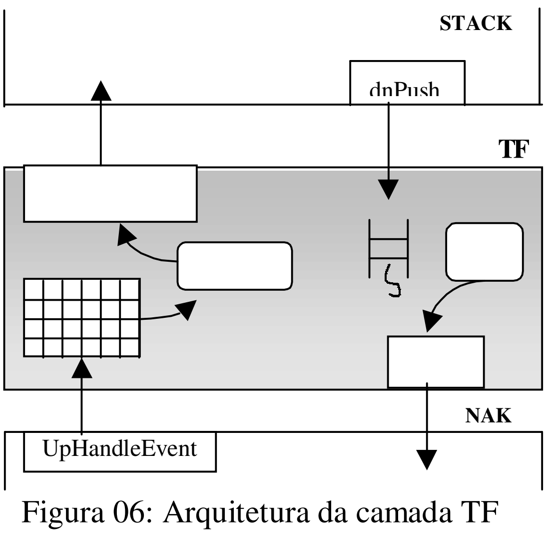

Figura 06: Arquitetura da camada TF

[4] A definição de bloco completo pode ser vista em [MES93, MAC94].

Além de ordenação total, a camada TF implementa o protocolo de *membership atômico*. O protocolo de *membership atômico* é responsável por manter a consistência na visão dos membros dos grupos, quando há mudança na configuração do grupo. Esta propriedade garante que a mudança na composição do grupo (entrada, saída e falha de membros) sejam percebidas em uma ordem mutuamente consistente entre as réplicas. Cada objeto aplicação do grupo mantém um protocolo de *membership atômico* que executa as seguintes funções. Digamos um grupo G_1 formado pelos objetos O_i, O_j, O_k.

- Quando um objeto O_i recebe um evento Visão suspeitando da falha de um objeto O_k, então:
- O_i inicia um acordo em relação a suspeita de falha de O_k.
    - Se todos os membros operacionais concordarem com a falha do objeto O_k, este objeto é retirado do grupo.
    - Senão, O_i irá solicitar ao grupo as mensagens que ele não recebeu de O_k.

O protocolo de *membership atômico* entra em ação a partir de um evento Visão enviado pela camada REACH do iBusTF suspeitando da falha de algum membro. Essa Visão é recebida na camada TF através da operação *upHandleEvent*. Digamos que o objeto O_i suspeitou da falha do objeto O_k, O_i cria um evento de Suspeita da falha de O_k e a envia para o grupo através da operação *dnPush*: (O_i, *suspect,* O_k, *lastBn*). O_i é o objeto que está suspeitando da falha de outro objeto O_k, e *lastBn* é o número de bloco da última mensagem recebida pelo objeto O_i de O_k. O objeto O_i também armazena esta suspeita em um vetor denominado *mySuspect*. Quando um outro membro do grupo, digamos O_j, recebe este evento de suspeita através da operação *upHandleEvent*, ele irá verificar se possui uma mensagem de O_k com número de bloco maior que o recebido na mensagem de suspeita. Em caso afirmativo, este objeto irá refutar esta suspeita enviando para o grupo o evento (O_j, *refute,* O_k, *suspBn*), informando o número de bloco da última mensagem recebida por ele de O_k. Caso contrário, esta suspeita será armazenada em um vetor denominado *otherSuspect.* Este vetor armazena as suspeitas recebidas por O_j dos outros membros do grupo.

Se o objeto O_i receber um evento *Refute*, este irá recuperar as mensagens não recebidas de O_k. O objeto O_i envia então um evento de recuperação de mensagens ao grupo (O_i, *recovery,* O_k, *upBn*). Os outros objetos do grupo ao receber este evento, enviam as mensagens de O_k com número de bloco maior que *upBn*.

O acordo sobre a suspeita de falha de O_k irá ser alcançado quando todos os membros concordarem com a última mensagem enviada por ele, através da operação *Agreement*. Digamos que o objeto O_i foi o primeiro membro que suspeitou da falha de O_k, então O_i irá chegar ao acordo sobre a falha de O_k quando todos os objetos armazenados em *mySuspect* forem confirmados, isto é, para cada objeto armazenado em *mySuspect* também existir uma entrada deste objeto em *otherSuspect* para todos os membros do grupo. Se o acordo é alcançado, então o objeto suspeito é retirado do grupo através da operação *updateView*. O objeto O_i envia um evento de confirmação ao grupo informando que o acordo sobre a falha dos objetos suspeitos foi alcançado (O_i, *confirmed, detection*) e que o membro foi retirado do grupo. d*etection* é um vetor que contém os objetos confirmados como falhos após o acordo. Os mebros do grupo ao receberem o evento Confirmed, verificam se já realizaram o acordo e em caso do acordo ainda não ter sido realizado, estes chamam a operação *updateView* para atualizar sua composição do grupo

Essas alterações não modificaram a estrutura do iBus nem sua interface com a aplicação, permitindo que qualquer aplicação iBus utilize o iBusTF sem modificações na estrutura de seus programas.

3.3.2. Dados de Desempenho

Para testar o desempenho do iBusTF, utilizamos dois mecanismos de medição. O primeiro mecanismo, *Round Trip*, 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. O segundo experimento calcula o tempo para realização do acordo sobre a falha de um membro. Os testes foram realizados utilizando-se quatro PC`s 300Mhz, conectados por uma rede *Ethernet* 10Mbps, sistema operacional Windows NT 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 membros e para cada tamanho de grupo foram enviadas 100 mensagens. Distribuímos os objetos ao longo das quatro estações utilizadas para os testes. Este experimento foi realizado com o intuito de medir o *overhead* causado pelos mecanismos de TF e foi realizado tanto no ambiente original iBus como no ambiente Tolerante a Falhas iBusTF implementado por nós. O gráfico abaixo mostra a média obtida através do mecanismo de *Round Trip* no iBusTF em relação a variação do tamanho do grupo.

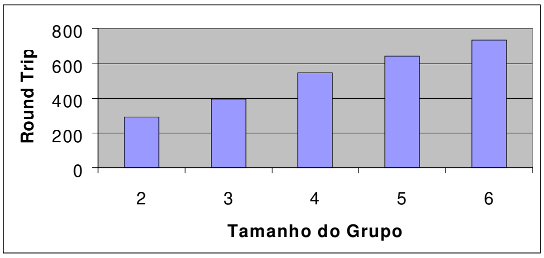

Figura 07: *Round Trip* no iBusTF.

Observamos a partir do gráfico da figura 07 que à medida que o tamanho do grupo aumenta, o tempo de *Round Trip* também aumenta. O gráfico da figura 08 apresenta os resultados obtidos (em *milisegundos*), a partir do mesmo experimento de *Round Trip* realizado no iBus original.

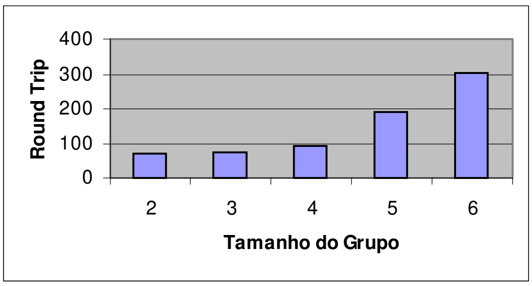

Figura 08: *Round Trip* no iBus

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.

O gráfico da figura 09 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. 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 *milisegundos*) é representado no gráfico abaixo.

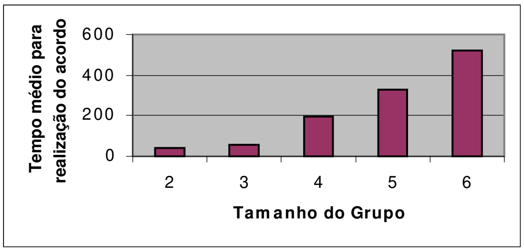

Figura 09: Gráfico do tempo médio para realização do acordo sobre a suspeita de falhas.

O tempo obtido é o resultado da média do acordo realizado por todos os objetos. Como observado no gráfico da figura 09, o tempo médio para realização do acordo também aumentou com a adição de novos membros ao grupo.

4. Trabalhos Correlatos

Outros trabalhos em andamento para Tolerância a falhas em JAVA são o Filterfresh, o ROAPI e o JavaGroups. O Filterfresh[CHR+99] é uma ferramenta para a construção de aplicações JAVA Tolerantes a Falhas baseada no *Remote Method Invocation* (RMI). 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 executada por outro objeto servidor, de forma transparente para o cliente. Da mesma forma que os objetos servidores, o Registrador do RMI também é replicado.

O ROAPI (*Replicated Object API*) [LEW99] é formado por um conjunto de classes JAVA, cujo objetivo principal, é oferecer uma API que permita a construção de objetos JAVA replicados. O ROAPI possui interface para os sistemas iBus e JAVAGroups [BAN98]. O JavaGroups, em desenvolvimento na Universidade de Cornell, é um ambiente para a construção de aplicações JAVA tolerantes a falhas, baseado no sistema Horus[RMB96]. A principal característica do JavaGroups em relação a outros Sistemas de Comunicação em

Grupo é a presença de *patterns*. Os *patterns* são classes JAVA localizadas abaixo do nível da aplicação utilizadas para facilitar o trabalho do programador da aplicação, visto que muitos dos recursos que seriam implementados a nível de aplicação, já encontram-se estruturados na forma de *patterns*.

Comparando o Filterfresh em relação ao iBusTF, este não oferece suporte à replicação de objetos clientes, apenas objetos servidores. Enquanto no Filterfresh cliente e servidor tem 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. O JavaGroups possui alguns serviços especificados (Transferência de Estado, Chamada Síncrona sobre Sistemas Assíncronos, etc.), que auxiliam o desenvolvimento de aplicação distribuídas, mas a maioria destes serviços não encontram-se ainda implementados.

Observamos que o ROAPI e o JavaGroups possuem interface para o iBus. 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.

5. Conclusão

Esse artigo apresentou uma abordagem para Tolerância a Falhas em JAVA através de Comunicação em Grupo. 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).

iBusTF 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.

Realizamos alguns testes de desempenho no iBusTF em quatro PC`s 300Mhz, conectados por uma rede *Ethernet* 10Mbps, sistema operacional Windows NT 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. 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, através de uma lista de discussão, em funcionamento desde agosto de 1998, e que permite aos programadores e usuários iBus trocarem informações sobre esse sistema.

Referências:

[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

[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.

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

[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.

[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.

[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.

[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.

[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.

[IONA95] Isis Distributed Systems In., IONA Technologies, LTD. *Orbix+Isis Programmer's Guide*, 1995. Document D070-00.

[LAM78] L. Lamport. *Time, Clocks and the Ordering of Events in a Distributed System*. Com. ACM, December, 1978, vol.36, n 12.

[LEW99] Scott Lewis. *ROAPI: Replicated Object API*. 1999. http://www.slewis.com/projects/README.html

[MAC94] Raimundo J. de A. Macêdo. *Fault Tolerant Group Communication Protocols for Asynchronous Systems*. PHD Thesis, August, 1994.

[MAC98] Raimundo J. de A. 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]*.

[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. *Newtop: a Total Order Multicast Protocol Using Causal Blocks*. First Year Report – Fundamental Concepts 1 of 3, BROADCAST ESPRIT Basic Research Project 6360, 1993.

[MES95] 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.

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

[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.

[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.

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