Relatório Técnico Interno

008/99

LaSiD

Comunicação em Grupo em Java: Adicionando Tolerância a Falhas ao iBus

Lilianne Dantas Cirne
Raimundo J. de A. Macêdo

Comunicação em Grupo em Java: Adicionando Tolerância a Falhas ao iBus

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—

Este artigo apresenta uma proposta para introduzir Comunicação em Grupo à linguagem Java. Os grupos são usados em Sistemas Distribuídos para prover tolerância a falhas através da replicação de componentes. A abordagem utilizada neste trabalho para suporte a Tolerância a Falhas foi incorporar protocolos fundamentais para Comunicação em Grupo, como os protocolos de Ordenação Total e Reconfiguração de Grupos à uma ferramenta, denominada iBus[MEM99], escrita totalmente em Java e voltada para a construção de aplicações *Intranet* como Cliente/Servidor e aplicações Multimídia.

Palavras Chaves— **protocolos de comunicação em grupo, ordenação total, tolerância a falhas, reconfiguração de grupos.**

I. INTRODUÇÃO

Com o advento dos Sistemas Distribuídos, o conceito de grupos tornou-se bastante comum nos dias atuais, alguns exemplos são as listas de discussão, onde um e-mail enviado por um usuário, é recebido por todos os membros inscritos numa lista. A propriedade fundamental dos grupos é a troca de mensagens através de difusão.

A abstração de grupos têm sido utilizada principalmente para a implementação de Tolerância a Falhas em sistemas distribuídos, cujo objetivo é manter a consistência do sistema, mesmo na ocorrência de falhas. As técnicas utilizadas para prover Tolerância a Falhas caracterizam-se pela redundância, ou seja, a presença de componentes (*Hardware e Software*) ou informações redundantes. Quando um componente falha, este poderá ser substituído por outro, não comprometendo a funcionamento do sistema como um todo. Além de serem utilizados para prover Tolerância a Falhas, os grupos possibilitam o aumento de desempenho e balanceamento de carga nos sistemas.

Apesar de existir um significativo número de sistemas que suportam Comunicação em Grupo (ISIS[BCJ$^+$90], Horus[RMB96], Phoenix[FG96], Newtop[EMS95], entre outros), a maioria dessas ferramentas não oferece suporte à linguagem Java. Nos últimos anos, Java vem adquirindo grande importância devido ao fato de ser portável, multitarefa, e possuir uma biblioteca de classes que facilitam a utilização de protocolos de rede amplamente utilizados como TCP/IP, HTTP e FTP.

Este artigo descreve uma proposta para incorporar a abstração de grupos com mecanismos de Tolerância a Falhas em uma ferramenta escrita puramente em Java, iBus[MEM99], desenvolvida com o objetivo de dar suporte a aplicações *Intranet*, como aplicações Cliente-Servidor e aplicações Multimídia. A escolha pelo iBus foi em função das vantagens oferecidas. O iBus apresenta uma estrutura em camadas de protocolos que possibilita às aplicações montarem as camadas de acordo com suas necessidades. Além disso, o iBus provê algum suporte básico a grupos como inserção e remoção de membros, criação e destruição de grupos e detecção de falhas. Neste artigo são apresentadas as modificações necessárias ao iBus para a introdução dos protocolos de Ordenação Total e Reconfiguração de Grupos. Também são apresentados resultados de performance, obtidos a partir deste novo ambiente, iBusTF (iBus Tolerante a Falhas), ou seja, a ferramenta iBus com adição de Tolerância a Falhas. Os protocolos de ordenação total e reconfiguração de grupos foram baseados no modelo proposto na BCG (Base Confiável de Comunicação em Grupo)[GM98] desenvolvida no LaSiD/UFBA.

O texto encontra-se estruturado da seguinte forma: a seção 2 apresenta o conceito de Comunicação em Grupo, descrevendo seus mecanismos necessários, e são exemplificadas algumas plataformas que provêm Comunicação em Grupo. Na seção 3 é apresentada a proposta deste trabalho para estender Java com Comunicação em Grupo, através da ferramenta iBus. Na seção 4 o iBusTF é comparado com outras plataformas de Comunicação em Grupo, na seção 5 são mostrados dados

---

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

de desempenho, e por fim na seção 6, a conclusão deste trabalho.

II. COMUNICAÇÃO EM GRUPO

Os grupos são usados em Sistemas Distribuídos para prover tolerância a falhas, através da replicação de componentes. Além disso, os grupos também permitem balanceamento de carga e aumento de performance, entre outras vantagens. Entretanto, todos os benefícios dos grupos só podem ser aproveitados se a criação e coordenação desses grupos forem bem estabelecidas.

Algumas propriedades devem ser observadas no desenvolvimento de *softwares* para implementar grupos, como o suporte para comunicação em grupo, incluindo endereço de grupo, atomicidade na entrega de mensagens, protocolos de ordenação de mensagens, e *Membership*. Essas propriedades encontram-se resumidas no tópico a seguir.

A. Propriedades fundamentais da Comunicação em Grupo

Endereço de Grupo: O endereço de grupo é usado para identificar de forma única uma coleção de membros.

Atomicidade: A propriedade de atomicidade garante que uma mensagem enviada a um grupo ou será recebida por todos os membros deste grupo, ou não será recebida por nenhum membro.

Reconfiguração (***Membership***): A propriedade de *membership* corresponde ao gerenciamento dos membros de um grupo. Este gerenciamento engloba as seguintes funções: 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*).

Ordenação: Para se obter o comportamento correto dos grupos, é necessário sincronizar a ordem em que as ações são executadas. Na **ordenação total**, todos os processos de um mesmo grupo recebem o mesmo conjunto de mensagens na mesma ordem. Na **ordenação causal**, apenas as mensagens não concorrentes são ordenadas[BIR93].

Sistemas e plataformas de Grupos: Os sistemas que tratam com Comunicação em Grupo incorporam a maioria das propriedades descritas anteriormente e em geral, adicionam outras funcionalidades inerentes a cada um. Há diversos sistemas que oferecem Comunicação em Grupo, como o ISIS[BCJ+90], Horus[RMB96], BCG[GM98], Phoenix[FG96], entre outros. A plataforma de Comunicação em Grupo BCG, serviu de modelo para a implementação dos protocolos de Ordenação Total e Reconfiguração de Grupos em Java que foram adicionados ao iBus.

III. ESTENDENDO O JAVA/IBUS COM TOLERÂNCIA A FALHAS

Java é uma linguagem Orientada a Objetos que tem sido bastante utilizada nos últimos tempos devido principalmente a sua portabilidade. A portabilidade do Java é conseguida através de uma linguagem intermediária denominada *Java bytecodes*[CAY97], obtida após a compilação de um programa Java. As instruções em *bytecodes* são executadas em qualquer sistema que possua a plataforma Java. A portabilidade é fundamental para integrar ambientes distintos, oriundos de diferentes fabricantes, principalmente com o advento da Internet.

Java também é Orientada a Objetos, simples, foi baseada em C++, e multitarefa. Apesar das vantagens oferecidas pelo Java, esta linguagem ainda não provê suporte à Comunicação em Grupo. Descreveremos neste trabalho uma forma de adicionar Tolerância a Falhas ao Java através da extensão de uma ferramenta totalmente escrita em Java, denominada iBus. O iBus, projetado por Maffeis[MAF96], tem como objetivo dar suporte a aplicações *Intranet* como *Groupware*, aplicações Cliente-Servidor e aplicações Multimídia. O iBus permite que aplicações Java enviem e recebam mensagens através de uma arquitetura estruturada em camadas de protocolos. Essa arquitetura permite à aplicação montar a sua pilha de protocolos de forma bastante flexível, a depender das suas necessidades (figura 1). Cada protocolo da pilha é responsável por uma função específica, como reenvio de mensagens perdidas (camada NAK), fragmentação de mensagens (camada FRAG), ordenação fifo (camada FIFO), detecção de falhas (camada REACH), e envio de dados (camada IPMCAST), entre outros. A camada STACK, localizada no topo da pilha é a interface entre a aplicação e o iBus. A versão do iBus 0.3 utilizada nesta implementação suporta os protocolos citados acima.

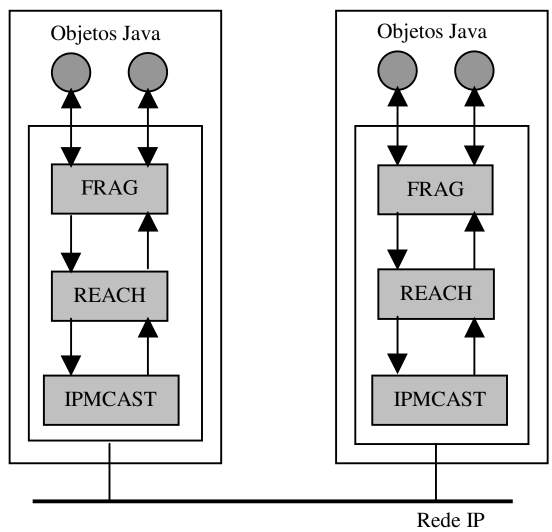

Figura 01: Arquitetura do iBus

A figura 1 mostra aplicações Java se comunicando através da pilha de camadas do iBus. Os eventos são enviados de uma camada a outra na pilha, e trafegam nos dois sentidos. Tods as mensagens que trafegam pelas camadas do iBus são denominados eventos. Os eventos são utilizados para distinguir mensagens enviadas da aplicação, das mensagens enviadas pelas camadas. Por exemplo, *evMessage*, mensagem enviada pela aplicação, *evAck*, mensagem enviada pela camada NAK para confirmação do recebimento das mensagens enviadas e *evView*, mensagem de mudança de visão enviada pela camada REACH, entre outros eventos. As camadas implementam protocolos com funções específicas, e podem modificar os eventos, acrescentando novas informações, criar novos eventos, descartá-los, ou simplesmente envia-los para as outras camadas sem modificações. A camada FRAG é responsável por fragmentar mensagens de tamanho maior que um valor predefinido. A camada REACH implementa um detetor de falhas baseado em *timeouts* e a camada IPMCAST é responsável por enviar os dados à rede.

O iBus permite a comunicação entre dois objetos ou entre um conjunto de objetos. A troca de mensagens entre os objetos de um grupo é feita através de canais. Um canal é identificado por um endereço *URL (Uniform Resource Locator)* utilizado para mapeamento dentro de um *Multicast* IP ou de uma conexão UDP ponto-a-ponto. Os objetos que se registram num mesmo canal fazem parte do mesmo grupo e recebem o mesmo conjunto de mensagens.

Neste trabalho incorporamos ao iBus uma nova camada denominada TF, onde foram implementados os protocolos de Ordenação Total e Reconfiguração de Grupos, necessários para Comunicação em Grupo Tolerante a Falhas. Esta nova versão do iBus com adição de Tolerância a Falhas denominaremos de iBusTF.

A. *Ordenação Total*

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 é utilizada uma estrutura de dados denominada Matriz de Blocos[MES94][EMS95], onde são guardadas as informações referentes à ordenação das mensagens enviadas e recebidas no grupo. Cada processo membro do grupo possui uma Matriz de Blocos local, formada por blocos causais. Bloco Causal é um vetor numerado seqüencialmente, de tamanho igual ao tamanho do grupo. A figura 02 apresenta uma Matriz de Blocos de um membro de um grupo composto por seis processos e com 5 blocos causais. Deste modo temos que, as linhas da matriz representam os blocos causais e as colunas identificam os processos que participam do grupo.

Quando um processo, digamos p_i, envia ou recebe uma mensagem com número de bloco b, caso este bloco causal b ainda não exista na sua matriz, ele criará mais uma entrada na matriz para este bloco causal e assinalará na coluna do processo p_i que a mensagem foi enviada. Supondo que a matriz de blocos da figura 02 representa a matriz de blocos do processo p_1 e que p_1 envia uma mensagem com número de bloco 5, este envio de mensagem será registrado na linha 5 coluna 1 da matriz. O recebimento de mensagens é registrado de forma semelhante na Matriz de Blocos. Na figura 02 as últimas mensagens recebidas dos processos p_2, p_3, p_4, p_5 e p_6 são respectivamente as mensagens com números de blocos 5,3,3,4 e 2.

	P_1	P_2	P_3	P_4	P_5	P_6
1	+			+		
2		+			+	+
3	+		+	+		
4	+				+	
5		+				

Fig.02 Uma Matriz de Blocos com seis processos, membros de um mesmo grupo.

A entrega das mensagens à aplicação é implementada através de uma *thread* denominada *Delivery* (figura 03). Esta *thread* verifica se o bloco causal satisfaz a condição de "bloco completo", e em caso afirmativo, as mensagens deste bloco são entregues em uma ordem preestabelecida. Essa ordem é necessária para garantir que as mensagens serão entregues em uma mesma ordem para todos os membros de um grupo.

Um bloco causal b é dito completo quando nenhuma mensagem poderá ser enviada ou recebida com número de bloco menor ou igual a b. Os blocos 1 e 2 da figura 2 estão completos, visto que todos os processos do grupo enviaram mensagens com número de bloco maior ou igual a 2. A condição de bloco completo é verificada através de um vetor denominado LRV (*Last Received Vector*), mantido localmente por cada processo. Este vetor registra as últimas mensagens enviadas e recebidas pelos processos. Cada vez que uma mensagem é enviada ou recebida, tanto o LRV quanto a Matriz de Blocos são atualizados. Podemos deduzir a partir da figura 02 que o vetor do processo p_1 é LRV=(4,5,3,3,4,2), verifica-se que os blocos causais 1 e 2 estão completos e existem 3 blocos incompletos.

Em algumas situações um processo pode permanecer por um longo intervalo de tempo sem enviar mensagens. Isto pode fazer com que um bloco não se complete. Para solucionar este problema, foi implementada uma *thread* denominada LTS (*Local Time Silence*), responsável por enviar mensagens nulas com o objetivo de completar blocos. Estas mensagens nulas contendo apenas informações de cabeçalho, são atualizadas na Matriz de Blocos e no LRV, mas não são entregues à aplicação.

Uma mensagem enviada pela aplicação passa primeiramente pela camada *Stack*, que é a camada de interface entre a aplicação e o iBus, e desta para a camada TF através do método *dnPush* (figura 03). Na camada TF essa mensagem é desempacotada e são adicionadas novas informações ao cabeçalho da mensagem, como um número associado a esta mensagem (número de bloco[1]), e o identificador do processo que está enviando esta mensagem. Este identificador é necessário para localizar o processo na Matriz de Blocos. No caminho contrário, as mensagens são enviadas da camada inferior à camada TF através de uma chamada ao método *upHandleEvent*. As mensagens são então armazenadas na Matriz de Blocos e posteriormente entregues à aplicação.

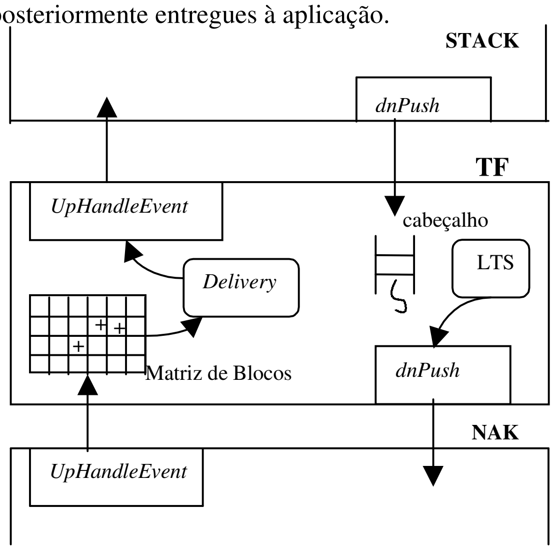

Fig.03 Arquitetura da Camada TF

Para introduzir a camada TF na pilha de camadas do iBus, implementamos os métodos necessários para integração desta camada ao ambiente iBus, *dnInit*, *dnPush*, *dnSubscribe*, *dnUnsubscribe* e *UpHandleEven*t. No método *dnInit* a camada TF é "inicializada" juntamente com as outras estruturas: vetores, Matriz de Blocos e *threads.* Os métodos *dnSubscribe* e *dnUnsubscribe* são responsáveis pela entrada e saída de membros, respectivamente. O método *dnpush* é usado para enviar os eventos de cima para baixo da pilha de protocolos. Neste método, adicionamos mais informações à mensagem, como um número de bloco associado a cada mensagem e o identificador do processo que está enviando a mensagem. Esse identificador único é necessário para identificar o processo em todas as Matrizes de Blocos. O método *upHandleEvent* é responsável por transmitir os eventos que chegam da rede, até a aplicação. As mensagens antes de serem entregues à aplicação, são armazenadas na Matriz de Blocos.

[1] Esse valor corresponde ao valor do relógio lógico de Lamport[LAM93].

B. Reconfiguração de Grupos

O protocolo de Reconfiguração de Grupos é responsável por manter a consistência dos membros dos grupos quando há mudança na configuração (entrada, saída ou falha de membros). Este protocolo é necessário para garantir que as mensagens em um grupo sejam entregues à aplicação, pois se um processo p_k falha, ele não poderá enviar mensagens e consequentemente os blocos causais da Matriz de Blocos não serão completados. Para garantir o funcionamento correto do protocolo de Ordenação Total, cada processo operacional mantém um protocolo de reconfiguração de grupos que executa as seguintes funções:

Um processo p_i recebe uma mensagem de suspeita da falha de um processo p_k.

Nesse caso, p_i inicia um consenso em relação a suspeita de p_k.

- Se todos os membros operacionais concordarem com a falha do processo p_k, é criada uma nova visão excluindo o processo falho.
- Senão, p_i irá solicitar ao grupo as mensagens que ele não recebeu de p_k.

O protocolo de reconfiguração de grupos entra em ação a partir de um evento visão (*evView*) enviado pela camada REACH do iBus. Esta camada implementa um protocolo de detecção de falhas baseado em *timeouts.* Os processos enviam mensagens periódicas ao grupo informando que estão operacionais. Se um processo p_k permanece um tempo sem enviar mensagens, os outros processos irão suspeitar da falha de p_k. A camada REACH de cada processo que suspeitou de p_k cria um evento denominado visão temporária, informando a suspeita deste processo. Esta visão temporária é então enviada às camadas superiores até atingir a camada TF.

Na camada TF de um dos processos que suspeitou da falha, digamos o processo p_i, esta informação de suspeita será armazenada em um vetor local denominado *my_suspect*. Este vetor armazena as suspeitas enviadas pela camada REACH de p_i. O processo p_i cria então um evento de suspeita de p_k, e o envia para o grupo (*p_i, suspect, pk, last_bn*), onde p_i é o processo que está suspeitando da falha de outro processo p_k, e *last_bn* é o número de bloco da última mensagem recebida pelo processo p_i de p_k. Quando um outro membro do grupo recebe este evento de suspeita, ele irá verificar se ele possui uma mensagem de p_k com número de bloco maior que o recebido na mensagem de suspeita. Em caso afirmativo, este processo irá refutar esta suspeita enviando o evento (*p_j, refute, pk, susp_bn*), informando o número de bloco da última mensagem recebida por ele de p_k. Caso contrário, esta suspeita será armazenada em um vetor denominado *other_suspect.* Este vetor armazena as suspeitas recebidas dos outros membros do grupo.

Se o processo p_i receber um evento *Refute*, este irá recuperar as mensagens não recebidas de p_k. O processo p_i envia então um evento de recuperação de mensagens ao grupo (*p_i, recovery, pk, up_bn*). Os outros processos do grupo ao receber este evento, enviam as mensagens de p_k com número de bloco maior que *up_bn*.

O consenso sobre a suspeita de falha de p_k irá ser alcançado quando todos os membros concordarem com a última mensagem enviada por ele. Digamos que o processo p_i foi o primeiro membro que suspeitou da falha de p_k, então pi irá alcançar o consenso quando todos os processos armazenados em *my_suspect* forem confirmados, isto é, para cada processo armazenado em *my_suspect* também existe uma entrada deste processo em *other_suspect* para todos os membros do grupo. Se o consenso é alcançado então é instalada uma nova visão retirando o processo suspeito do grupo, e atualizando a Matriz de Blocos e o LRV. O processo p_i envia um evento de confirmação ao grupo informando que o consenso foi alcançado (*p_i, confirmed, detection*). d*etection* é um vetor que contém os processos confirmados como falhos após o consenso.

Os membros do grupo ao receberem este evento, verificam se o vetor *detection* é um subconjunto do seu vetor de processos suspeitos, *mysuspect.* Em caso afirmativo, este processo ainda não realizou o consenso. Ele então atualizará *mysuspect* (*mysuspect= mysuspect – detection*) e instalará a nova visão.

Os protocolos de Ordenação Total e de Reconfiguração de Grupos seguiram a especificação do iBus versão 0.3. Implementamos os métodos necessários (*dnPush, upHandleEvent, dnInit*, entre outros) para que a camada TF pudesse ser incluída na pilha de camadas do iBus e acrescentamos os outros métodos responsáveis pelo funcionamento correto dos protocolos implementados, como as *threads Delivery* e LTS e os eventos *Suspect*, *Recovery*, *Confirmed* e R*efute*.

IV. COMPARAÇÃO COM OUTRAS ABORDAGENS

Outra maneira de estender o paradigma de Comunicação em Grupo ao Java é utilizar a linguagem Java para ter acesso a um serviço de Comunicação em Grupo já implementado em alguma plataforma, como a plataforma CORBA[OMG95]. Assim, não seria necessário implementar novas classe, apenas fazer uma chamada ao serviço de Comunicação em grupo oferecido pela plataforma CORBA. A desvantagem desta abordagem é que o serviço de Comunicação em Grupo para CORBA ainda não está padronizado.

Outra ferramenta também escrita totalmente em Java para Comunicação em Grupo, é o *JavaGroups*[BAN98]. Esta ferramenta difere do iBus pela utilização de *patterns*. Os *patterns* são usados para facilitar o trabalho do programador da aplicação, visto que muitos dos recursos que seriam implementados na aplicação, já estão estruturados em uma camada intermediária, onde encontram-se os *patterns*. No entanto, esta ferramenta ainda está em fase de implementação na universidade de Cornell, e, assim como o iBusTF, utiliza o sistema iBus para prover uma camada de transporte.

V. DADOS DE DESEMPENHO

Para testar o desempenho dos novos protocolos adicionados ao iBus, utilizamos dois mecanismos de medição. O primeiro, o mecanismo de Round Trip, calcula o tempo que um processo leva para enviar uma mensagem a um grupo e receber a resposta de confirmação (*ack*) de um dos membros. O tamanho do grupo variou de 2 a 6 membros e para cada tamanho de grupo foram enviadas 100 mensagens. O gráfico abaixo mostra a média obtida em relação a variação do tamanho do grupo. Os testes foram realizados utilizando-se quatro PC`s 300Mhz, conectados por uma rede *Ethernet* 10Mbps e sistema operacional Windows95.

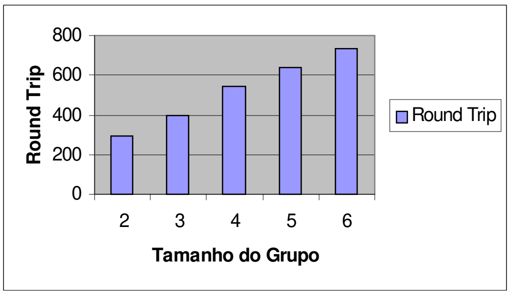

Fig.04 Escalabilidade segundo R*ound Trip.*

A medida que o tamanho do grupo aumenta, o tempo de *Round Trip* também aumenta. Os valores variaram de 245 a 760 milisegundos.

O segundo gráfico mostra o tempo para realização do consenso. O tempo médio de consenso é a diferença entre o recebimento de uma mensagem de visão enviada pela camada REACH do iBus, suspeitando da saída de um membro, e a completa remoção deste membro do grupo. O tempo obtido é o resultado da média do consenso realizado por todos os processos. Neste experimento, foram realizados testes com grupos de 2 a 6 processos e onde apenas um processo falha durante cada execução. O tempo médio para realização do consenso é representado no gráfico abaixo.

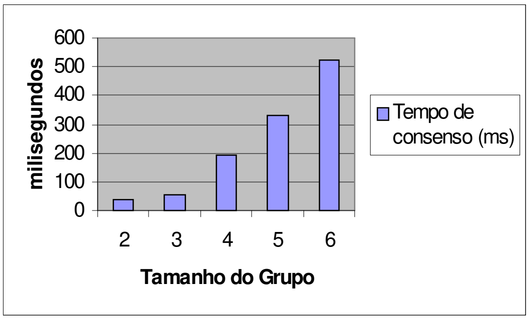

Fig.05 Gráfico do tempo médio para realização do consenso

O tempo médio de consenso também aumentou com o tamanho do grupo, visto que mais comparações devem ser feitas para atingir o consenso.

VI. CONCLUSÃO

Neste artigo foi apresentada uma forma de estender Java com Comunicação em Grupo, baseado no iBus. iBus, desenvolvido por Maffeis[MAF96], é uma ferramenta escrita totalmente em Java com o objetivo de dar suporte a aplicações *Intranet* como *Groupware* e aplicações multimídia. O iBus possui uma arquitetura estruturada em camadas de protocolos e um suporte básico a grupos. Nosso trabalho adicionou uma nova camada ao iBus denominada TF, onde foram implementados os protocolos de Ordenação Total e Reconfiguração de Grupos, protocolos estes imprescindíveis para Comunicação em Grupo Tolerante a Falha.

Para analisar o desempenho dos protocolos implementados na camada TF, realizamos dois experimentos. O primeiro experimento, denominado *Round Trip*, calcula o tempo que um processo leva para enviar uma mensagem ao um grupo e receber uma resposta de um dos membros. O tamanho do grupo variou de 2 a 6 membros, e a medida que o tamanho do grupo foi acrescido, o tempo de *Round Trip* também foi aumentado. O outro experimento calcula o tempo para realização do consenso sobre a falha de um membro. Os grupos também variaram de 2 a 6 membros, e o tempo de consenso também aumentou proporcionalmente com o tamanho do grupo. Testes de *Rond Trip* e Consenso sob os mesmos protocolos, mas desenvolvidos em C++ são encontrados em [GM98]. Os valores obtidos neste artigo são maiores comparados com a implementação em C++. Este fato é justificado por Java ser interpretada. A execução de aplicações escritas em linguagem interpretadas é mais lenta que linguagens não interpretadas.

Melhorias ainda podem ser acrescentadas a este trabalho no sentido de que um objeto possa pertencer a de mais de um grupo, e permitir a inserção de novos membros ao grupo dinamicamente. Em [MES94] é descrito em detalhes o algoritmo para inserção de novos membros. A implementação descrita neste artigo ainda não suporta esta facilidade.

REFERÊNCIAS

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

[EMS95]Paul Ezhilcevan, Raimundo A Macêdo, Santosh K Shrivastava. "*Newtop: a Faul-Tolerant Group Communication Protocol*". 15th International Conference on Distributed Computing Systems. Vancouver - Canada. June, 1995. IEEE.

[BIR93]Kenneth P Birman. "*The process Group Approach to Reliable Distributed Computing*". Communications of ACM. December, 1993.

[LAM93] L. Lamport. *Time, Clocks and the Ordering of Events in a Distributed System*. Com. ACM, December, 1993, vol.36, n 12.

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

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

[MES94]R. J. Macêdo. *Fault Tolerant Group Communication Protocols for Asynchronous Systems*. Tese de Doutorado, agosto de 1994.

[BAN98]Bela Ban. *Design and Implementation of a Reliable Group Communication Toolkit for Java.* Dept. of Computer Science. Cornell University. December 1998.

[MAF96]Silvano Maffeis. *iBus – Java Intranet Software Bus* (http://www.softwired.ch/people/maffeis). 1996

[CAY97]Gary Cornell, Cay S. Horstmann. *Core Java.* Makron Books, 1997.

[ROD98]Robert Orfali, Dan Harkey. *ClientSever Programming with Java and CORBA.* Second Edition.1998

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

[MEM99] Altherr Marcel, Martin Erzrberger and Silvano Maffeis. *iBus – A Software Bus for the Java Plataform*. Java Report. September, 1999.

[GM98] Fabíola Greve, Raimundo Macêdo. *The BCG Membership Service Performance Analysis*. SBRC98. Rio de Janeiro. Maio, 1998