Universidade Federal da Paraíba

Centro de Ciências e Tecnologia

Curso de Mestrado em Informática

Lições Aprendidas no Projeto e Implementação de uma Ferramenta de Comunicação em Grupo Baseado no Paradigma de Canais de Eventos

MARCOS XAVIER DE ALMEIDA BARRETTO

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal da Paraíba como requisito parcial para a obtenção do grau de Mestre em Informática .

Prof. Dr. Francisco Vilar Brasileiro

(orientador)

Campina Grande

MAIO – 2003

BARRETTO, Marcos Xavier de A.

B273L.

Lições Aprendidas no Projeto e Implementação de uma Ferramenta de Comunicação em Grupo Baseado no Paradigma de Canais de Eventos

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

145 p. Il.

Orientador: Dr. Francisco Vilar Brasileiro

1. Sistemas distribuídos.
2. Engenharia de software.
3. Ferramentas de Comunicação em Grupo.
4. Modelos de Estruturação para Protocolos de Comunicação Especializados Baseado em Canais de Eventos.
5. Modelos de Estruturação para Protocolos de Comunicação Especializados Baseado em Camadas.

CDU - 681.3.066D

Lições Aprendidas no Projeto e Implementação de uma Ferramenta de Comunicação em Grupo Baseado no Paradigma de Canais de Eventos

Marcos Xavier de Almeida Barretto

Dissertação aprovada em ___ /___/___

Prof. Dr. Francisco Vilar Brasileiro
(orientador)

Dr. Álvaro Francisco C. Medeiros

Dr. Walfredo Cirne

Agradecimentos

Agradeço a todos que me ajudaram, que acreditaram em mim e que me fizeram chegar até aqui.

Agradeço especialmente ao meu orientador Francisco Vilar Brasileiro por me mostrar o caminho a trilhar.

Agradeço a minha esposa e filha por serem a minha motivação maior.

Agradeço aos meus Pais por me darem a oportunidade de andar com minhas próprias pernas.

Agradeço a meus irmãos, em especial a Márcio, pelo companheirismo e pelos preciosos conselhos nas horas difíceis.

Agradeço aos colegas de trabalho pela sua paciência e compreensão.

Agradeço especialmente a Josilávio e Denise pela compreensão e apoio fundamentais para a conclusão dos meus trabalhos.

Agradeço aos colegas de mestrado, especialmente a Mário, pelo ombro amigo na hora de dividir as aflições relativas ao mestrado.

Finalmente, agradeço a todos os meus professores que através da sua experiência me ajudaram a enxergar um pouco além.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Gráficos

Resumo

Este trabalho discute quais as vantagens e desvantagens de se construir protocolos de comunicação de alto nível, utilizando o modelo baseado em canais de eventos como forma de estruturação interna. É efetuado um estudo comparativo teórico e experimental com o modelo baseado em camadas. Este modelo de estruturação interna é o mais difundido entre os protocolos de comunicação especializados existentes.

Inicialmente, são apresentados os conceitos inerentes a cada modelo estudado, são discutidas as suas principais virtudes e problemas. Em seguida, apresentamos as principais funcionalidades do iBusTF e do iBusTFE, que são os protocolos de comunicação especializado, mais especificamente ferramentas de comunicação de grupo (FCG), utilizados para fazer a comparação experimental entre os dois modelos de estruturação. Ressaltamos que o iBusTFE foi construído durante esta pesquisa especialmente com esta finalidade, assim ele possui as mesmas funcionalidades do iBusTF, porém utiliza o modelo baseado em canais de eventos como forma de estruturação interna, ao invés do modelo baseado em camadas.

Apresentamos a seguir, a metodologia utilizada para comparar os dois modelos estudados e quais as medidas adotadas para eliminar e/ou minimizar a influência de fatores adversos nos testes. Finalmente, apresentamos os dados obtidos nos testes e fazemos a análise dos dados buscando verificar se as hipóteses levantadas na discussão conceitual foram confirmadas ou não. Apresentamos então, algumas considerações sobre diversos fatores que influenciaram significativamente os resultados dos testes com as duas FCGs.

Abstract

This work discusses the advantages and disadvantages of building high-level communication protocols using the model based on event channels as internal structural form. It is realized a theoretic and experimental comparative study with the layer based model, that is the most spread out between the specialized communication protocols that exist.

At the beginning, we present the concepts of each studied model, discussing their main virtues and problems. To follow we present the functionalities of the iBusTF and iBusTFE, that are the specialized communication protocols, more specifically group communication tools (GCT), used to make the experimental comparison between both of the structural models. We stand out that the iBusTFE was built with this purpose, and therefore he has the same functionalities of the iBusTF, however, the iBusTFE uses the model based on event channels as its internal structural form, instead of the layer based model.

Then we present the methodology used to compare both of the studied models and which were the adopted measures to eliminate what the influence of adverse factor on the tests. Finally, we present the data that was obtained at the tests and we make the its analysis, trying to verify if the hypotheses raised at the conceptual discussion were confirmed or not.

1 Introdução

A distribuição do processamento em diversas máquinas, interligadas por uma rede de comunicação, tem se tornado cada dia mais comum no desenvolvimento de sistemas que têm como requisitos a escalabilidade, confiabilidade, performance e boa relação entre custo e capacidade de processamento. Estes sistemas são compostos por diversos processos que interagem entre si através do uso de protocolos de comunicação.

Os protocolos de comunicação são responsáveis em fornecer os serviços de comunicação requeridos pelas aplicações distribuídas [VKCD99], encapsulando a complexidade inerente à transmissão de dados entre os processos que compõem o sistema em questão, permitindo assim, que as aplicações se especializem no negócio.

Por conseguinte, há aplicações distribuídas com os mais diversos propósitos que requerem serviços de comunicação que podem variar desde serviços simples, como o envio de mensagens ponto-a-ponto não confiável até serviços complexos como a ordenação total, visão de grupo atômica, comunicação de um-para-muitos (*multicast*) etc.

Existem diversos protocolos de baixo nível altamente difundidos no mercado (ex.: TCP/IP [COM98, TAN96]). Eles provêem serviços de comunicação básicos, utilizados por grande parte dos sistemas. Porém, devido à grande diversidade de requisitos, freqüentemente protocolos especializados [BGH$^+$b00, CM00, MGH98, MMSA+96, RBH94, MES93] são requeridos pelas aplicações distribuídas.

Tipicamente os protocolos especializados são construídos utilizando como base os protocolos de mais baixo nível [BGH$^+$b00, CM00, MGH98, MMSA+96, RBH94, MES93]. As necessidades específicas são agregadas aos serviços disponibilizados pelos protocolos de baixo nível, formando o conjunto específico de serviços requeridos. Podemos citar as ferramentas de comunicação de grupo (FCG) como bons exemplos de sistemas especializados em prover serviços de comunicação de alto nível como: comunicação *multicast*, ordenação total, visão atômica de grupo entre outros [BGH$^+$b00, CM00, MGH98, MMSA+96, RBH94, MES93].

Historicamente os protocolos de comunicação têm sido projetados e construídos com base no modelo em camadas. Podemos citar vários exemplos como: TCP/IP [COM98 TAN96], ISO/OSI [TAN96], Horus [RBH94] etc. Este modelo surgiu como uma alternativa para contornar as limitações do modelo plano[1], principalmente no que se refere à manutenibilidade e validação.

Os protocolos de comunicação construídos com base no modelo em camadas são compostos por diversas camadas de software que são dispostas logicamente uma sobre a outra, criando uma pilha.

Este modelo provê um esquema de funcionamento extremamente simples, que permite que os protocolos de alto nível sejam decompostos em camadas coesas. Elas fornecem funcionalidades específicas que quando utilizadas em conjunto compõem os serviços de comunicação requeridos.

O modelo em camadas introduziu uma série de benefícios muito importantes impactando diretamente a flexibilidade, manutenibilidade e reusabilidade dos protocolos de comunicação baseados em camadas. Estes benefícios fizeram com que este seja o modelo mais utilizado nos protocolos de comunicação existentes.

Porém, algumas restrições como a sobrecarga de tráfego, a ausência de paralelismo e o consumo excessivo de recursos são impostas por este modelo.

Apesar de haver estudos visando minimizar os problemas encontrados, eles propõem técnicas que conseguem minimizar os problemas do modelo em camadas, mas não conseguem elimina-los por completo. As técnicas de otimização propostas impõem um preço elevado, apesar de serem eficientes no que diz respeito à melhoria da performance. Elas restringem a flexibilidade, manutenibilidade e reusabilidade, principais virtudes do modelo em camadas.

Uma alternativa para contornar os problemas impostos pelo modelo em camadas é a utilização do modelo baseado na abstração de canais de eventos [OPSS93, BGH+a00]. Com ele, as aplicações encapsulam os dados a serem enviados a outros processos em eventos. Não existem camadas, mas entidades

[1] A criação do modelo plano pode ser considerada como a primeira tentativa para construir protocolos de comunicação estruturados. Sua principal virtude é o excelente desempenho, porém a manutenibilidade e validação, em sistemas complexos, podem se tornar tarefas extremamente complicadas.

que podem ser produtoras e/ou consumidoras de eventos. Para enviar os dados a aplicação gera eventos e os encaminha para um canal de eventos que é responsável por gerenciar a entrega às entidades que registrem interesse em consumi-los.

As entidades fornecem funcionalidades específicas, de forma similar às camadas do modelo anterior. Porém, a comunicação entre as entidades é gerenciada pelos canais de eventos que possuem uma lista de todos os produtores e consumidores de eventos. Portanto, um mesmo tipo de evento pode ser gerado/consumido por diversas entidades, bastando que estas se registrem junto ao canal de eventos para a produção e/ou consumo de eventos.

Desta forma, as vantagens do modelo em camadas apresentadas anteriormente são preservadas e suas restrições são resolvidas ou minimizadas [OPSS93, BGH+a00].

Diversos estudos foram realizados apresentando o modelo baseado em canais de eventos e suas vantagens [OPSS93, BGH+a00, BGH+b00], adicionalmente, alguns sistemas foram construídos utilizando este paradigma [OPSS93, BGH+b00]. Porém, há uma escassez de estudos que comprovem ou refutem as hipóteses levantadas pelos autores que defendem a utilização do modelo baseado em canais de eventos, principalmente no que se refere à comparação com o modelo em camadas.

A contribuição deste trabalho é avaliar o quanto a utilização do modelo de canais de eventos pode ajudar no projeto e implementação de protocolos de comunicação de alto nível. O objetivo é demonstrar as vantagens e desvantagens do modelo em camadas e do baseado em canais de eventos, através de um estudo de caso, trazendo assim mais luz para a discussão sobre a utilização do modelo baseado em canais de eventos. Para isso, foi adotada uma abordagem experimental. Foi construído o iBusTFE que é uma ferramenta de comunicação de grupo (FCG) [BBb03] baseada no modelo de canais de eventos. Ela foi baseada em uma FCG existente chamada iBusTF [CM00], construída seguindo o modelo baseado em camadas. Ambas FCGs disponibilizam os mesmos serviços para as aplicações. A visão atômica do grupo e a ordenação total de mensagens são os principais serviços implementados.

Para atingir o objetivo traçado, nossa preocupação foi a de isolar as variáveis que poderiam afetar a comparação, mantendo o foco da análise, na

diferença estrutural entre os modelos. A linguagem utilizada, as funcionalidades, o ambiente de testes, entre outros fatores foram controlados para que a comparação se concentrasse na diferença entre os paradigmas sobre os quais as FCGs foram construídas.

O critério utilizado para comparação entre as duas FCGs foi o desempenho, ou seja, a performance medida em unidades de tempo para enviar e receber dados entre processos remotos.

Para obter os dados comparativos foram efetuados diversos testes utilizando uma pequena aplicação geradora de consensos [GHO^{+}00], construída durante esta pesquisa exclusivamente para este fim, a qual utiliza o iBusTF ou o iBusTFE.

A obtenção dos dados sobre o desempenho foi efetuada com o auxílio de uma ferramenta de perfilamento (Quantify – Rational [RAT02]). Assim, foi possível analisar não somente os tempos para obtenção dos consensos, mas também o tempo de execução de cada método, dos seus métodos descendentes e a quantidade de chamadas, entre outros fatores estudados e demonstrados neste trabalho.

Desta forma, verificamos quais as principais virtudes e limitações das duas FCGs, identificando falhas de projeto, erros de implementação e principalmente as restrições impostas pelo modelo em que se basearam, que são o foco principal deste trabalho.

As limitações encontradas são analisadas de forma comparativa neste trabalho. Boas práticas para o projeto e a implementação de protocolos de comunicação são sugeridas, visando maximizar os resultados com cada um dos modelos estudados. Finalmente, quando possível, são indicadas técnicas de otimização para contornar os problemas impostos pelos dois paradigmas.

Com base nos testes, na análise dos dados coletados e na constatação das limitações de cada modelo realizamos uma análise visando destacar qual modelo demonstrou um melhor resultado no caso estudado.

O restante dessa dissertação está organizado da seguinte forma. No Capítulo 2 é feita uma descrição a respeito dos modelos utilizados para estruturação de protocolos de comunicação. Nos capítulos 3 e 4 o projeto e a implementação do iBusTF e do iBusTFE são discutidos em detalhes, respectivamente. A metodologia aplicada para a obtenção dos dados e a

avaliação do desempenho das duas FCGs são descritas no Capítulo 5, e finalmente no Capítulo 6 são apresentadas as conclusões e sugestões para trabalhos futuros.

2 Modelos para Estruturação de Protocolos de Comunicação

2.1 Introdução

O estudo dos modelos de estruturação é de grande importância na construção de protocolos de comunicação. As vantagens e desvantagens inseridas por cada modelo têm papel de fundamental importância, pois interferem diretamente sobre itens como a flexibilidade, manutenabilidade, reusabilidade e performance. Estes itens se apresentam como requisitos muito comuns entre as aplicações distribuídas.

Os protocolos de comunicação têm a função de fornecer serviços de comunicação entre processos onde há a ausência de memória compartilhada, uma situação típica de sistemas distribuídos. A comunicação entre processos deste tipo é efetuada através da troca de mensagens. O processo remetente cria uma mensagem e a envia para o protocolo de comunicação indicando qual o processo destinatário e que tipo serviço de comunicação deseja. O protocolo de comunicação envia a mensagem ao processo destinatário fazendo uma chamada ao sistema operacional que é responsável por remeter a mensagem através da rede de comunicação[2]. No computador remoto que executa o processo destinatário, o sistema operacional entrega a mensagem ao protocolo de comunicação que por sua vez entrega os dados ao processo remetente em concordância com o tipo de serviço de comunicação solicitado [TAN95]. A Figura 2.1 descreve o esquema de envio de mensagens entre processos remotos discutido neste parágrafo.

[2] Alguns protocolos de baixo nível podem ser implementados no próprio sistema operacional ou mesmo no hardware (p.e.: camada ethernet do protocolo TCP/IP).

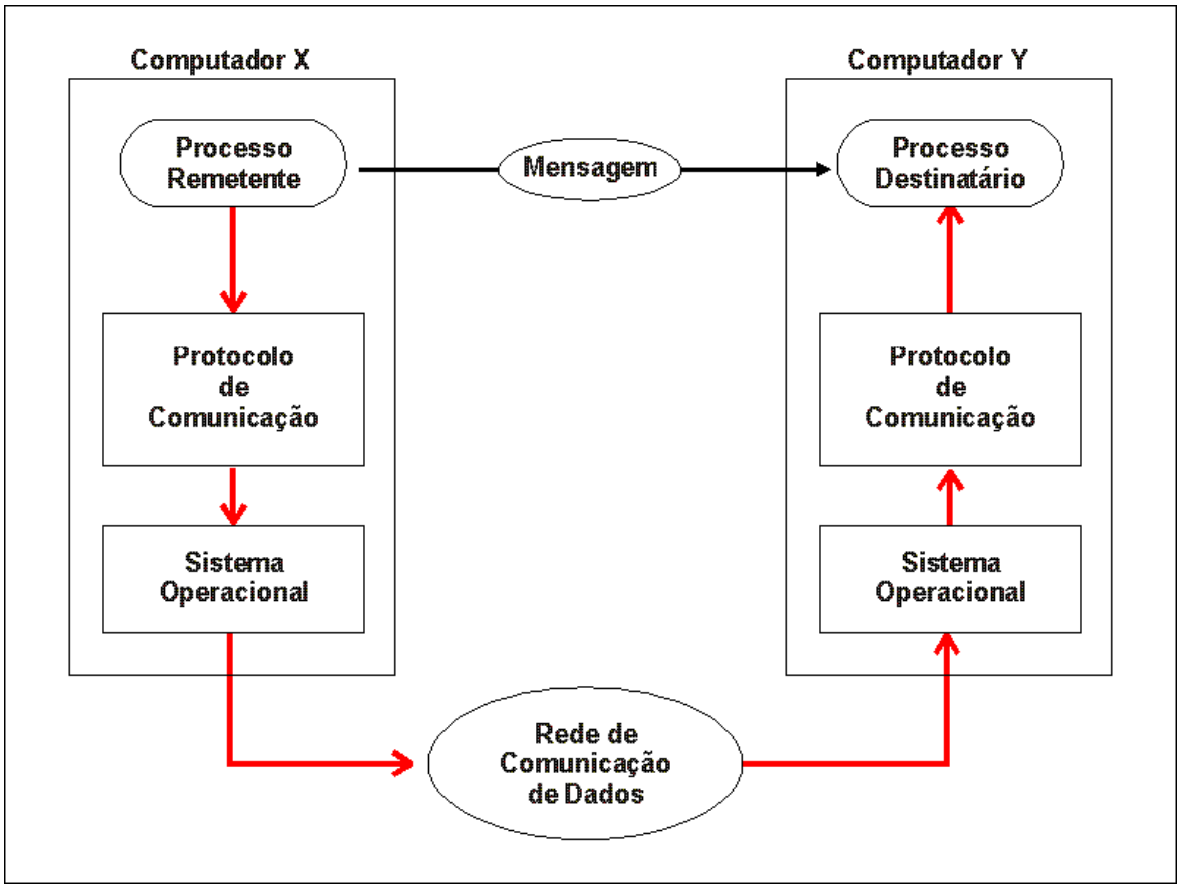

Figura 2.1 - Comunicação entre processos através do envio de mensagens

Com base no modo em que são estruturadas internamente, podemos dividir os protocolos de comunicação em três grandes grupos: **planos, em camadas e baseados em eventos**.

Os protocolos planos foram os primeiros a surgir, seguidos dos em camadas e finalmente pelos baseadas em eventos. Os modelos mais novos foram propostos com base na experiência dos modelos anteriores. Desta forma, eles almejam solucionar ou ao menos minimizar os problemas dos modelos de construção existentes previamente.

Todos os três tipos defendem a divisão dos sistemas em módulos. A forma de interação entre os módulos é justamente o principal diferencial entre os três modelos aqui citados. Cabe fazer uma ressalva, que a modularização de sistemas surgiu com o objetivo de estruturar internamente sistemas complexos, para que eles pudessem ser construídos e mantidos com maior facilidade, fato que incontestavelmente permitiu uma grande evolução no desenvolvimento de sistemas. Porém, o fato é que historicamente a modularização é um dos fatores que mais influencia negativamente o desempenho de um sistema, pois insere

retardos devido à interação entre os módulos. Isto nos leva a um dilema entre escolher um programa com boa performance ou com boa estruturação. Os modelos aqui apresentados buscam manter e melhorar a forma de estruturação interna, minimizando ao máximo o preço a ser pago por isso, porém é importante frisar que será pago um preço.

O modelo plano propõe que os protocolos sejam decompostos em módulos, visando segmentar sistemas de grande complexidade em pequenas porções de fácil entendimento e construção. Não existem regras para determinar como a comunicação entre os módulos deve ocorrer. Há diversos exemplos de FCGs deste tipo, podemos citar o Ensemble [MGH98], o BCG [MAC95,GM98] e o Totem [MMSA+96].

Os protocolos construídos com base neste modelo possuem como principal virtude o excelente desempenho. Porém, apresentam diversos problemas como (1) a dificuldade para incluir/excluir serviços, (2) a dificuldade de validação e (3) entendimento difícil gerado por uma cadeia extensa, e muitas vezes confusa, de chamadas de funções entre diversos módulos de forma encadeada. Em função das desvantagens apresentadas, o modelo plano não vem sendo utilizado na construção de novos protocolos de comunicação[3].

Outro modelo estudado é o baseado em camadas, o mais utilizado pelos protocolos de comunicação existentes. Ele oferece um esquema extremamente simples, com regras claras que facilitam a construção de novos sistemas, oferece boa manutenibilidade e permite que a validação dos sistemas seja executada facilmente. Porém, problemas com a geração de sobrecarga de tráfego (*overhead*), tratamento serial de mensagens e processamento desnecessário de mensagens por algumas camadas são freqüentemente relatados em estudos sobre este modelo [HR96, BGH$^+$a00].

Com o intuito de resolver os problemas do modelo baseado em camadas, surgiu o modelo baseado em canais de eventos. Além de tratar especificamente os problemas que interferem na performance dos protocolos de comunicação, este modelo oferece outras vantagens, como por exemplo, o total desacoplamento de tempo, espaço e fluxo entre os diversos módulos que interagem entre si.

---

[3] Está fora do escopo deste trabalho uma discussão detalhada sobre o modelo plano.

O capítulo 2 tem como objetivo apresentar qual a forma de funcionamento do modelo baseado em camadas e do modelo baseado em eventos, ressaltando as virtudes e identificando quais as restrições impostas pelos dois modelos citados. Desta forma, este capítulo contribui para a dissertação fornecendo o embasamento teórico, ou seja, as hipóteses que deverão ser comprovadas ou refutadas nos capítulos seguintes através de uma abordagem experimental.

2.2 Modelo Baseado em Camadas

O modelo baseado em camadas surgiu como uma opção para resolver os principais problemas do modelo plano. Uma das suas principais características é a de ter um fluxo claro por onde os dados são conduzidos obrigatoriamente para que a comunicação entre processos seja estabelecida. Este fato facilita a validação do sistema como um todo.

Os protocolos de comunicação construídos com base neste modelo são compostos por diversas camadas de software que são dispostas logicamente uma sobre a outra, criando uma pilha.

As aplicações que utilizam estes protocolos se comunicam exclusivamente com a camada que se encontra no topo da pilha. Através desta camada a aplicação envia e recebe dados de outros processos.

Para haver comunicação entre dois processos distintos, os dados são recebidos pela camada que está no topo da pilha de camadas. Daí cada camada faz o tratamento necessário para o envio da mensagem, agregando cabeçalhos de controle e repassando os dados e os cabeçalhos agregados para as camadas inferiores da pilha.

No envio de dados, as camadas apenas se comunicam com as camadas adjacentes de nível inferior, esta regra não se aplica à última camada da pilha. Ela recebe os dados e faz uma chamada ao sistema operacional que remete os dados através da rede de comunicação para o processo remoto. O sistema operacional do computador remoto recebe os dados da rede de comunicação e os entrega ao processo destinatário, para a camada de mais baixo nível. Neste instante, os dados recebidos trafegam no sentido inverso. Eles são enviados para cima na pilha de camadas até que atinjam a camada que está no topo da pilha. Ela é responsável por entregar os dados para a aplicação de destino.

Cada camada ao receber os dados lê o cabeçalho de controle que foi agregado pela camada de mesmo nível no processo remetente e o exclui, repassando para as camadas superiores apenas os dados.

Como o fluxo dos dados é bem definido, o modelo em camadas consegue evitar a “teia de chamadas” a métodos de diferentes módulos que criavam sérios problemas no modelo plano. Com isso, este modelo viabilizou um grande avanço no que se refere à manutenabilidade e à validação, quando comparado ao modelo plano.

Entre um par de camadas adjacentes é definida uma interface que determina quais são os serviços que a camada inferior presta à camada superior, e como estes serviços devem ser acessados. Estas interfaces são de fundamental importância, pois facilitam muito os testes e a manutenção das camadas isoladamente, permitindo que novas funcionalidades sejam facilmente agregadas ou até excluídas, caso seja necessário.

Cada camada é responsável por disponibilizar serviços que são utilizados pelas camadas superiores para prover novas funcionalidades. Quando agregadas, as funcionalidades das camadas inferiores permitem que a camada que se encontra no topo da pilha possa disponibilizar para a aplicação os serviços por ela requeridos.

Logicamente, uma camada de nível “n” apenas se comunica com camadas do mesmo nível em outros processos. As camadas do processo remetente agregam informações de controle que são utilizadas pelas camadas de mesmo nível nos processos destinatários para prover as funcionalidades definidas na interface existente com a camada superior. As informações de controle, organizadas em cabeçalhos, não são repassadas para as camadas de nível superior.

Desta forma, podemos afirmar que o modelo em camadas segue o princípio da divisão de camadas [COM98] que determina que uma camada “n” de destino deverá receber exatamente o objeto enviado pela camada “n” de origem. As mensagens geradas pelas camadas são compostas pelos dados recebidos das camadas superiores e as informações de controle geradas pela própria camada. Este princípio assegura que ao desenvolver sistemas em camadas os desenvolvedores apenas necessitarão se preocupar com a camada que estão trabalhando, sem se preocupar com o comportamento das camadas inferiores.

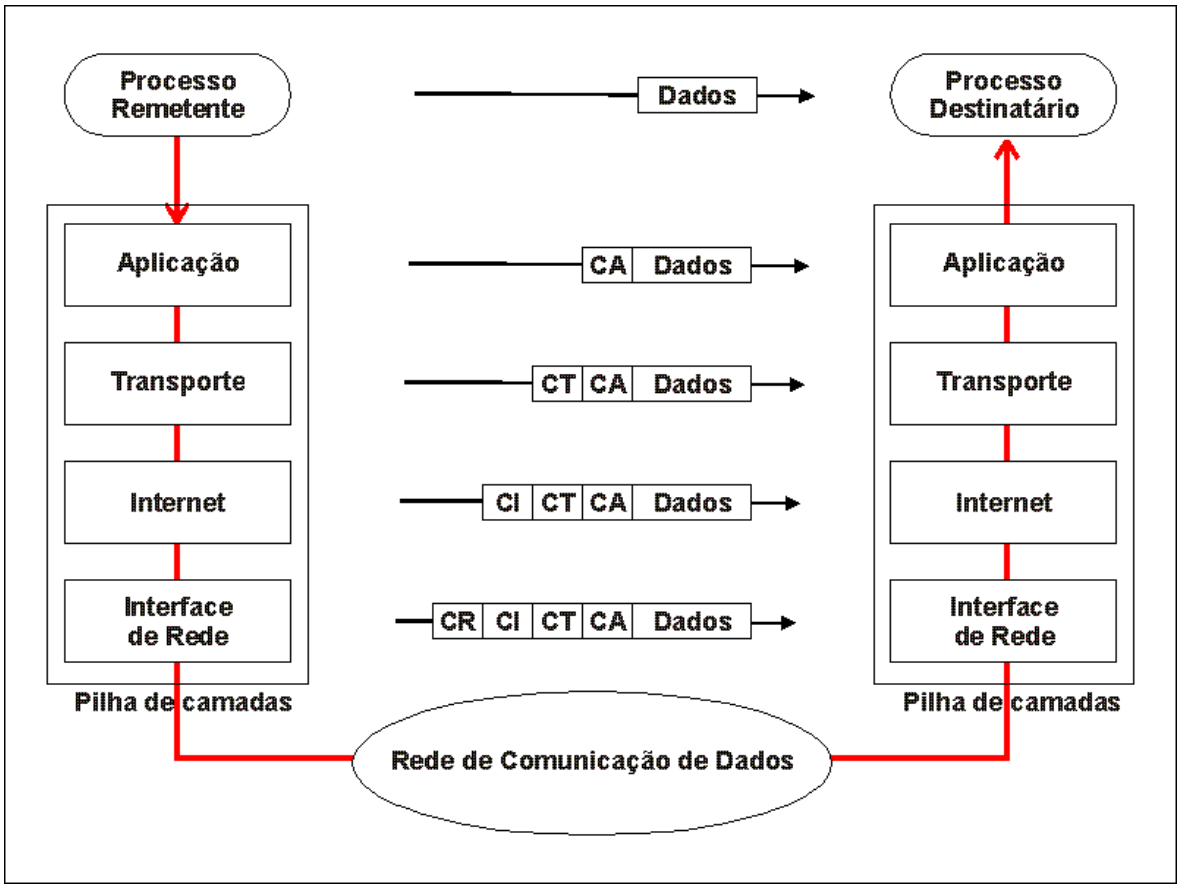

Figura 2.2 - Envio de dados entre processos utilizando o protocolo TCP/IP que é baseado em camadas.

Na Figura 2.2 podemos observar como a comunicação em protocolos baseados em camadas é realizada. Os dados recebidos da aplicação são repassados pela pilha de camadas. Todas as camadas processam e agregam cabeçalhos com informações de controle à mensagem antes de repassar a mensagem para a camada adjacente inferior. No processo destinatário as camadas recebem exatamente o mesmo objeto que foi gerado pela camada de mesmo nível no processo de origem. Isto ocorre porque cada camada utiliza e posteriormente remove apenas as informações de controle que lhe cabem, repassando o restante da mensagem para que as camadas superiores tratem. Assim, fica clara a utilização do princípio da divisão de camadas citado anteriormente.

Esta abordagem introduz uma série de vantagens em relação aos protocolos de comunicação planos muito difundidos no passado [HR96, BGH+a00]:

- Facilidade para testar as camadas isoladamente.

- Facilidade para testar a pilha de camadas como um todo, devido ao fluxo dos dados bem definido que pode ser observado com facilidade.
- A inclusão de novas funcionalidades é facilmente realizada através da construção de uma ou mais camadas.
- Melhoria da reusabilidade do código, pois as camadas existentes podem ser combinadas de outras formas, bastando adaptar-se às interfaces disponibilizadas pelas camadas adjacentes.
- As camadas podem ser substituídas facilmente, através da criação de novas camadas que implementem as mesmas interfaces.

Estas vantagens trouxeram benefícios muito importantes otimizando a flexibilidade, a manutenabilidade e a reusabilidade dos protocolos de comunicação baseados em camadas, fazendo com que este seja o modelo mais utilizado nos protocolos de comunicação existentes.

O esquema extremamente simples de funcionamento do modelo em camadas permite que protocolos de mais alto nível sejam construídos baseando-se em protocolos de baixo nível. A segmentação dos serviços a serem implementados em camadas permite que novos protocolos especializados sejam criados apenas adicionando-se novas camadas à pilha existente.

Porém, o modelo em questão impõe algumas restrições que afetam principalmente o desempenho dos sistemas [HR96, BGH+a00]:

- **Sobrecarga de tráfego** – As camadas agregam informações de controle às mensagens enviadas pela aplicação no processo remetente e após o envio pela rede, no processo destinatário. Estas informações são utilizadas pela camada de mesmo nível na pilha de protocolos. Estes dados de controle não são gerados ou consumidos pela aplicação, porém, são enviados pela rede de comunicação. Como não são estes dados não possuem utilidade explícita para a aplicação são considerados como sobrecarga de tráfego. Quanto maior o número de funcionalidades (camadas), maior será a sobrecarga imposta.
- **Ausência de paralelismo** – O modelo em camadas por concepção prevê o tratamento serializado dos dados. As camadas ao receber os dados fazem o processamento necessário e então os enviam para cima

ou para baixo na pilha de camadas. Conseqüentemente, duas camadas distintas não podem tratar os dados em questão concomitantemente.

- **Consumo excessivo de recursos** – O modelo impõe o tratamento compulsório de todos os dados enviados entre dois processos por todas as camadas da pilha. Em alguns casos determinadas camadas não precisariam tratar os dados, pois a funcionalidade que agregam não tem relação direta com os dados em questão. Porém, devido ao esquema imposto pelo modelo todas as camadas necessitam tratar os dados, gerando em alguns casos um consumo desnecessário de recursos.

Diversos estudos visando minimizar os problemas encontrados foram realizados e várias técnicas de otimização propostas. Efetivamente elas conseguem diminuir a sobrecarga utilizando técnicas como a compactação de cabeçalhos, identificação e serialização dos fluxos de mensagens e alteração na ordem do processamento das camadas [VKCD99].

Apesar de minimizarem as restrições relativas à sobrecarga e tratamento desnecessário de mensagens estas técnicas criam outros problemas. Elas quebram algumas regras do modelo em camadas, como a comunicação exclusiva entre camadas adjacentes, com isto, há um obscurecimento da codificação e conseqüentemente uma maior dificuldade em executar a validação. Outra grande desvantagem é a redução da produtividade, pois estas técnicas não possuem formas automáticas de execução.

Em suma, as técnicas de otimização propostas, apesar de serem eficientes no que diz respeito à melhoria da performance, impõem um preço elevado. Elas restringem a flexibilidade, a manutenabilidade e a reusabilidade, principais virtudes do modelo em camadas.

2.3 Modelo Baseado em Canais de Eventos

O modelo baseado em canais de eventos é totalmente orientado a objetos e tem como objetivo oferecer um novo paradigma de comunicação entre componentes de software. Seu ponto forte é a comunicação totalmente assíncrona entre os diferentes componentes de software. Ele foi proposto no início dos anos 90 [OPSS93] e tem sido utilizado como uma alternativa ao modelo

baseado em camadas, para resolver ou ao menos minimizar os problemas existentes.

Apesar de ter sido proposto originalmente e muito utilizado com o intuito de interligar processos remotos [OPSS93, EFGK01, Ske98], cenário típico de sistemas distribuídos, o paradigma de comunicação apresentado pode ser aplicado em outros contextos. Porém, sempre com o objetivo de prover a comunicação entre componentes de software diversos, seja entre processos distintos ou dentro de um mesmo processo.

O funcionamento de sistemas construídos com base neste modelo é fundamentado basicamente em três abstrações: entidades, eventos e canais de eventos.

Entidades – Equivalem no modelo plano aos módulos e no modelo em camadas às próprias camadas. São utilizadas pelo projetista com o objetivo de segmentar o problema a ser resolvido e devem prover funcionalidades específicas. A granularidade, ou seja, o número de funcionalidades providas por cada entidade é uma decisão que o projetista deve tomar[4].

Os serviços prestados pelo sistema em construção são obtidos através da interação entre as diversas entidades. Elas se comunicam através da troca de objetos chamados de eventos. Uma entidade pode ser produtora e/ou consumidora de eventos.

Eventos – São objetos que encapsulam os dados a serem trocados pelas entidades, podem conter qualquer tipo de dados. Existem eventos especialmente construídos para integrar processos remotos, neste caso os dados ou atributos dos eventos, devem ser serializáveis para que possam de transmitidos pela rede de comunicação de dados.

Canais de Eventos – Oferecem uma espécie de serviço de entrega de eventos, ou seja, servem como mediadores entre as entidades produtoras e as consumidoras de eventos provendo a comunicação entre elas. As entidades consumidoras registram seu interesse em consumir determinados tipos de

[4] Em qualquer sistema de informação modularizado, a regra geral a ser seguida é manter alta a coesão dentro dos módulos e baixo acoplamento entre eles. No que se refere ao acoplamento, o próprio modelo baseado em eventos resolve o problema, pois provê total desacoplamento entre as entidades. Desta forma, cabe ao projetista definir a coesão de cada entidade.

eventos junto aos canais que armazenam estes registros de interesse. Por sua vez, as entidades produtoras enviam os eventos produzidos para os canais. Assim, os eventos são recebidos pelos canais de eventos e com base nas tabelas armazenadas é entregue uma cópia a cada uma das entidades consumidoras que subscreveram interesse em consumi-los. O serviço de notificação de eventos é responsável por prover o armazenamento e gerencia das inscrições para consumo de eventos e a entrega propriamente dita dos eventos de forma eficiente.

A Figura 2.3 apresenta o esquema geral de interação entre as entidades, de acordo com o modelo baseado em canais de eventos. Para registrar seu interesse em eventos, as entidades consumidoras fazem uma chamada ao método **`subscribe()`**[5] do objeto canal de eventos. Os registros de intenção de consumo de eventos recebidos pelos canais de eventos são armazenados para uso posterior como pode ser notado na tabela de consumidores registrados no Canal de Eventos. Caso alguma entidade não tenha mais interesse em receber eventos de um tipo, esta deve invocar o método **`unsubscribe()`** do canal de eventos, que providenciará a remoção do registro de interesse armazenado previamente. Pode se observar na Figura 2.3, que a entidade “consumidor 1” no instante t2 solicitou a retirada de sua inscrição para consumir os eventos do tipo “1” e “2”, conseqüentemente no instante t3 a tabela de consumidores registrados aparece atualizada na figura.

Para produzir um evento, as entidades tipicamente invocam o método **`publish()`**, do canal de eventos, passando como parâmetro o próprio evento. Então, o canal envia uma cópia do evento recebido para cada entidade consumidora que registrou interesse por este tipo de evento através de uma chamada ao método **`notify()`**, disponibilizado pelos consumidores. Os produtores de eventos podem adicionalmente registrar quais tipos de eventos irão produzir chamando o método **`advertise()`**, a partir desta informação as entidades consumidoras podem tomar conhecimento sobre quais eventos estão disponíveis.

---

[5] O nomes de métodos utilizados nesta dissertação são meramente didáticos, variando de acordo com as diferentes implementações.

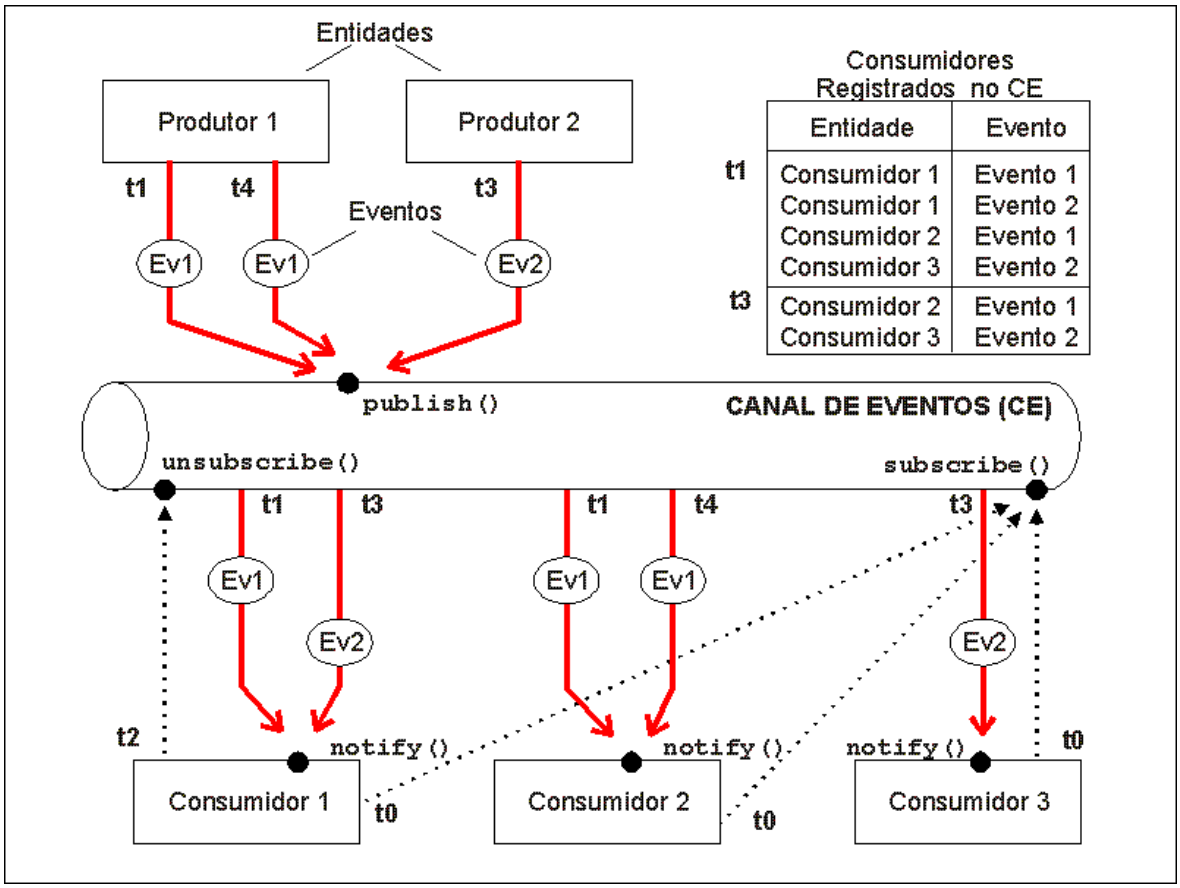

Figura 2.3 - Comunicação assíncrona entre entidades seguindo o modelo baseado em canais de eventos.

Analisando a comunicação entre as entidades podemos observar que aquelas que produzem e as que consomem os eventos estão totalmente desacopladas devido ao uso do canal de eventos. Elas se comunicam tipicamente de forma anônima e assíncrona, os consumidores recebem os eventos, porém não sabem que entidade os produziu. De forma similar, os produtores enviam os eventos produzidos para o canal de eventos e não sabem qual ou quais entidades que irão consumi-los.

Apesar da comunicação assíncrona ser o ponto forte do modelo baseado em canais de eventos, em alguns casos as aplicações necessitam se comunicar de forma síncrona. Este tipo de comunicação é amplamente utilizada nos sistemas distribuídos existentes (p.e.: Java RMI, CORBA, Microsoft DCOM etc.) sobre denominações diferentes, por exemplo, como modelo “requisição/resposta” ou “cliente/servidor”.

A comunicação síncrona exige que uma requisição seja enviada de uma entidade para outra, então a entidade de origem aguarda a resposta da entidade

de destino de forma síncrona. Simplificando, uma entidade necessita invocar um método de outra entidade passando os parâmetros requeridos e então aguardar o retorno com o valor resultante. Há algumas alternativas para implementar este tipo de comunicação no modelo baseado em eventos:

- Em [OPSS93] a solução foi quebrada em duas partes distintas: a) encontrar o servidor (entidade invocada) e b) estabelecer a conexão e invocar o método requerido. Portanto, com o intuito de descobrir o endereço do servidor, o cliente (entidade que invoca) envia um evento específico questionando quais os servidores disponíveis para prestar o serviço requerido. Em seguida, todos os servidores habilitados respondem com outro evento, indicando o seu endereço. Assim, o cliente pode receber vários endereços como resposta, ficando a seu critério escolher qual servidor deverá ser invocado. Outra alternativa proposta é haver alguma política entre os servidores para que apenas um responda.
- Em [BGH+a00] a solução proposta foi similar à adotada para a comunicação assíncrona. Nela o canal de eventos serve como mediador entre a entidade que faz a requisição e a que responde. Desta forma, as entidades que podem ser invocadas registram junto ao canal de eventos quais métodos estão disponíveis para a comunicação síncrona, aqui chamada de serviços. Na Figura 2.5 o registro do serviço em questão pode ser observado no instante t0 através da chamada ao método **`registerService().`** Todos os serviços possuem uma identificação única que é repassada pela entidade servidora quando registra o serviço junto ao canal de eventos. Os serviços disponíveis ficam registrados junto ao canal de eventos que guarda qual a entidade, qual o método a ser chamado na entidade (*handler)* e o identificador do serviço. Assim, quando um cliente necessitar de um serviço específico ele deve invocar o canal de eventos, passando o identificador do serviço requerido utilizando o método **`invokeRequest()`**. O canal de eventos será responsável por chamar o método da entidade servidora, registrado previamente, relativo ao serviço em questão,

passando os parâmetros requeridos. Ao receber a resposta a esta chamada o canal irá devolver como o retorno da chamada ao método **`invokeRequest()`** um objeto onde está contida a resposta à solicitação efetuada pela entidade cliente, fato que está apresentado no momento t2 da Figura 2.5. Dessa forma, as entidades permanecem totalmente desacopladas apesar da comunicação síncrona.

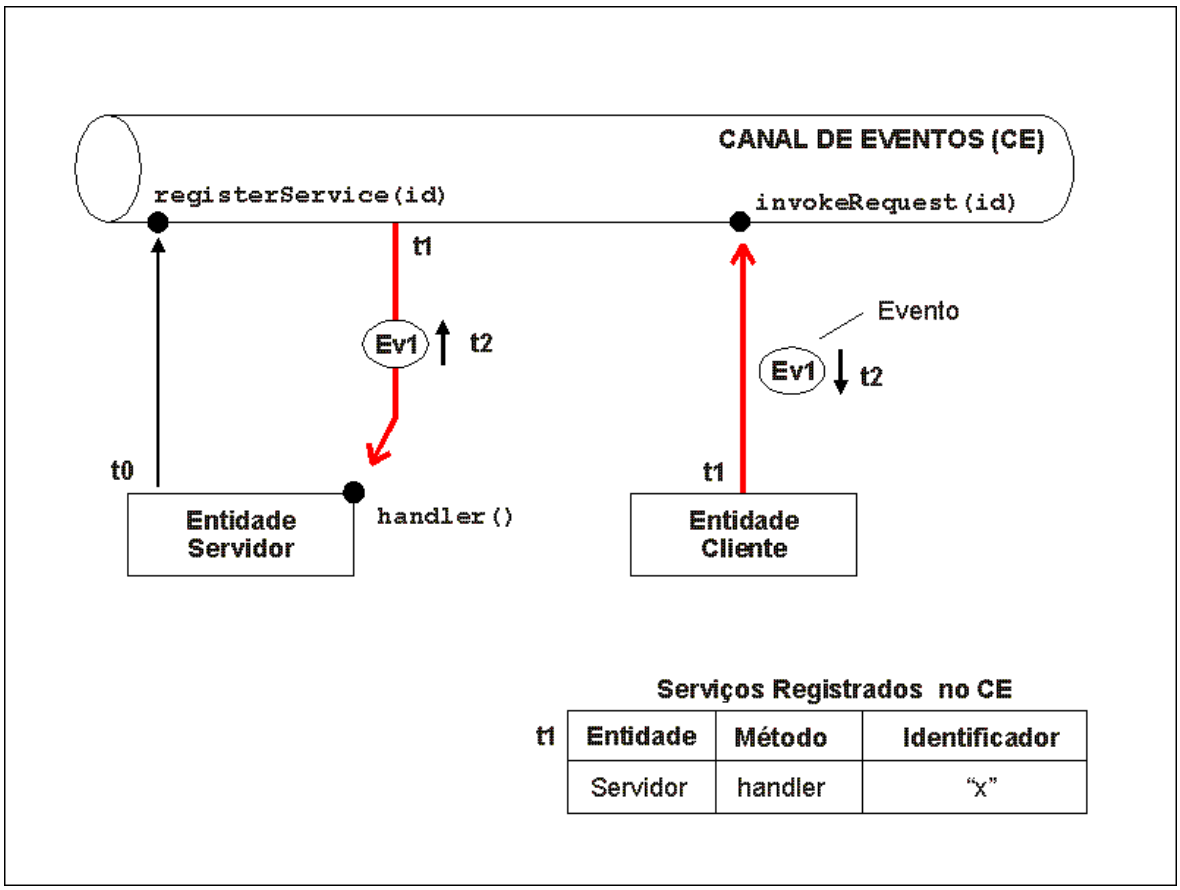

Figura 2.4 - Comunicação síncrona entre entidades seguindo o modelo baseado em canais de eventos.

É importante notar que independente da forma de comunicação ser síncrona ou assíncrona, o modelo baseado em canais de eventos provê total desacoplamento entre as entidades.

Podemos analisar o desacoplamento obtido pelo modelo de canal de eventos de três formas distintas:

Desacoplamento de espaço – É o tipo de desacoplamento mais facilmente percebido. Como o canal de eventos é o responsável por mediar a comunicação entre as entidades e elas não possuem nenhuma referência mútua,

não é necessário saber quantas ou quais entidades estão consumindo/produzindo eventos para que a comunicação seja estabelecida.

Desacoplamento de tempo – Os produtores e consumidores de um evento não necessitam obrigatoriamente estar ativos em um mesmo instante. Por exemplo, um produtor pode encaminhar um evento ao canal de eventos e logo em seguida ser encerrado. O fato de não estar mais ativo não irá influenciar no consumo do evento produzido por parte das entidades consumidoras.

Desacoplamento de fluxo – As entidades não precisam ser bloqueadas para enviarem/receberem eventos. Por exemplo, as entidades consumidoras de eventos não precisam verificar se algum evento foi recebido de forma cíclica (*polling*), pois ao ser produzido um evento elas serão notificadas pelo canal de eventos através da entrega do evento (*callback*) e os produtores não são bloqueados ao produzir eventos, apenas fazem uma chamada assíncrona a um método e prosseguem seu processamento normalmente. A produção e consumo de eventos não ocorrem no fluxo principal dos produtores e consumidores.

O desacoplamento existente no modelo baseado em canais de eventos permite que a disponibilidade, a escalabilidade, a manutenabilidade e a flexibilidade do sistema sejam maximizadas, pois removem quaisquer dependências entre as entidades permitindo que elas sejam facilmente substituídas, duplicadas e adaptadas para atender a novos requisitos. Em algumas implementações [OPSS93] a criação de novas entidades pode ser feita em tempo de execução, aumentando a disponibilidade do sistema.

Com relação à forma com que as entidades registram interesse para consumir eventos, há três variações dentro do modelo baseado em canais de eventos. Eis um resumo sobre cada uma delas:

Registro baseado em tópicos – Esta foi a forma originalmente proposta nos primeiros trabalhos que citam o modelo [OPSS93]. Nesta variante, cada evento é rotulado com um conjunto de caracteres que formam um tópico, um conceito parecido com o de grupo. Portanto, as entidades consumidoras se registram para consumir eventos relacionados a um tópico, ou seja, fazem parte do grupo de entidades que consomem um evento de um determinado tipo. Para facilitar o agrupamento de eventos algumas melhorias a esta forma de registro foram incorporadas com a utilização de tópicos hierarquizados. Assim, foram inseridos os conceitos de tópicos e sub-tópicos, onde uma entidade pode através

de um único registro para consumo receber todos os eventos relativos a um tópico e todos os seus sub-tópicos. Outro avanço foi a utilização de caracteres de pesquisa (“*wildcards*”) associados aos tópicos hierarquizados que oferecem a possibilidade de registro a diversos tópicos e sub-tópicos cujos nomes seguem um determinado padrão. Devido à criação de regras estáticas este tipo de registro oferece um bom desempenho, porém apresenta fortes limitações.

A divisão de eventos em tópicos é criticada, pois não é considerada natural [EGS01, EG01]. Desta forma, ela insere uma filtragem de eventos com granularidade muito alta, ou seja, a divisão em tópicos não permite que as aplicações consigam expressar todas as diferentes peculiaridades que desejam para consumir determinados tópicos. Uma alternativa para resolver este problema é a criação de um grande número de tópicos, porém esta abordagem dificulta a gerência dos tópicos, provocando envio de eventos redundantes e apresentaria limitações no que se refere à escalabilidade das implementações que utilizam IP *multicast* devido ao número limitado de endereços disponíveis [EGS01].

Registro baseado no conteúdo – Este tipo de registro permite que os eventos sejam escolhidos com base em suas propriedades, avaliadas em tempo de execução. Estas propriedades podem representar atributos internos do evento ou meta dados associados ao evento. Logo, as entidades podem construir regras complexas para registrar seu interesse em determinados eventos. Estas regras são baseadas em pares indicando a propriedade e o valor em que as entidades têm interesse (p.e.: propriedade = idade, valor > 15 anos). As regras estabelecidas são verificadas quando do recebimento dos eventos, em tempo de execução, e determinam se eles devem ser entregues às entidades. Existem algumas formas distintas para que as entidades registrem interesse no consumo de eventos baseando-se no conteúdo:

String – A entidade envia um conjunto de caracteres que correspondem à regra com os pares de propriedade e valor respectivos. Normalmente, estas Strings estão em conformidade com alguma linguagem de consulta padrão de mercado (ex.: SQL). É introduzido um problema ao utilizar as linguagens de consulta, pois os eventos são implementados em uma linguagem e as regras para registros expressas em outra, dificultando o registro de regras complexas, além de levar à quebra do encapsulamento dos objetos (eventos) através do acesso direto aos seus atributos.

Modelo de Objeto – Nesta forma de registro, as entidades fornecem um objeto que indica ao canal de eventos que todos os eventos com tipo igual ao do objeto indicado devem ser enviados para a entidade. Os atributos do objeto em questão podem ser preenchidos indicando que apenas os eventos que tenham atributos iguais aos indicados são de interesse. Os atributos com valores nulos não são avaliados. Esta forma de registro é limitada, pois não permite que sejam registrados intervalos de valores que se tenha interesse. Os valores dos atributos dos eventos em tempo de execução devem ser idênticos ao do objeto modelo.

Código executável – Ao se registrar junto ao canal de eventos as entidades consumidoras fornecem um objeto chamado de filtro, que tem a capacidade de avaliar em tempo de execução quais os eventos são de interesse e quais devem ser descartados. Portanto, o próprio objeto conhece o evento e seus atributos e implementa as regras que desejar. Esta forma de registro é a mais flexível, porém, ela fere o encapsulamento de objetos defendido na orientação a objetos, pois os filtros acessam diretamente os atributos dos eventos. Para contornar este problema foi sugerida uma variação neste tipo de abordagem que utiliza a reflexão para invocar métodos dos próprios eventos e comparar o resultado com valores pré-definidos [EG01]. Assim, os atributos não são acessados diretamente pelos filtros, mas por métodos definidos pelo próprio evento. Todavia, com o uso da reflexão os métodos a serem invocadas são descobertos em tempo de execução e este tipo de operação tem um custo de processamento alto, inserindo um retardo considerável, que faz com que este tipo de registro possua uma escalabilidade limitada [EG01, EFGK01, EGS01].

Na Figura 2.6 podemos observar o envio de eventos com a utilização de filtros, sem reflexão. No instante t0 a entidade “consumidor 2” se registra junto ao canal de eventos para consumir eventos do tipo “1”, passando um filtro como referência, note que a entidade consumidor ”1” se registra também, porém não passa nenhum filtro. No instante t1 o “Produtor 1” envia um evento do tipo requerido, as duas entidades registradas recebem o evento, porém antes do Canal de Eventos enviar o evento ao “Consumidor 2” o filtro é aplicado ao evento, como a regra nele estabelecida é satisfeita o registro é então enviado. Quando o “Produtor 2” envia um novo evento o Canal de Eventos aplica novamente o filtro registrado antes de enviar o evento ao “Consumidor 2”, neste caso a regra não é satisfeita, indicando que a entidade consumidora não tem interesse em consumir

o evento. Portanto, podemos observar que o evento do tipo “1” gerado no instante t2 somente é entregue à entidade “Consumidor 1”.

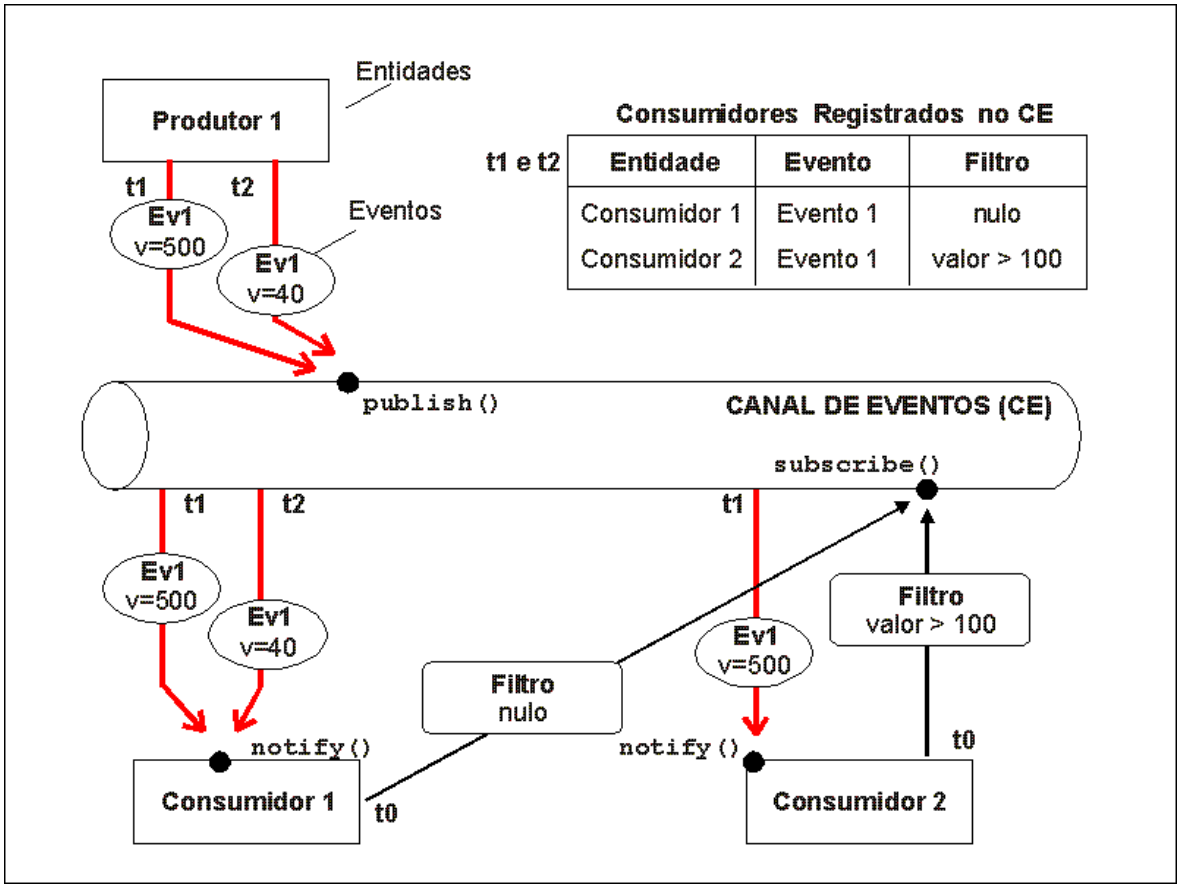

Figura 2.5 - Registro para consumo de eventos baseado no conteúdo através de código executável (filtro)

Registro baseado em tipos – Esta abordagem é conceitualmente similar ao registro baseado em tópicos. Se equipararmos os conceitos de tipo de eventos e tipo de objetos as duas formas de registro a eventos se tornam similares, pois um tópico está diretamente relacionado a um tipo de evento. Porém, no registro baseado em tipos ao invés de demonstrar interesse em tópicos as entidades registram interesse em tipos de objetos, no caso os eventos. Assim, é possível registrar interesse por um tipo de eventos e todos os tipos descendentes, de forma similar ao registro de um tópico e os seus sub-tópicos [EFGK01, EGS01]. Contudo, esta abordagem apresenta uma performance superior aos outros tipos de registro, pois a verificação do tipo de um determinado objeto é feita de forma muito eficiente pelas linguagens de programação existentes. Pode ser verificado em [EGS01], que esta abordagem é mais rápida do que agregar novos atributos

aos eventos e os verificar em tempo de execução. Como limitação a este método de registro podemos citar o fato que as diversas linguagens de programação lidam com a sub-tipagem de forma diferente. Assim, a noção de registro a um tipo e seus sub-tipos pode variar entre diversas linguagens existentes no mercado[6].

O desempenho é um ponto de fundamental importância para os sistemas distribuídos, algumas técnicas foram propostas para otimizar o desempenho dos sistemas estruturados de acordo com o modelo baseado em canais de eventos. Abaixo destacamos algumas das técnicas propostas:

Envio direto de eventos – Em sistemas onde o grupo de entidades ligadas a um canal de eventos é grande e a comunicação entre elas é muito intensa, fazendo com que um número elevado de eventos necessariamente seja processado, pode haver problemas de performance devido à concentração de um grande fluxo de dados a serem processados em um único ponto do sistema, mas especificamente no canal de eventos. Para contornar este problema pode ser utilizada uma técnica de otimização que permite aos produtores enviarem os eventos produzidos diretamente para as entidades consumidoras, sob a coordenação dos canais de eventos que ficam responsáveis por notificar os produtores quando houver alguma alteração no grupo de entidades consumidoras de eventos [BGH^{+}a00]. Para implementar este esquema de notificação, as entidades produtoras são notificadas pelo canal de eventos quando as entidades consumidoras de eventos registram interesse nos eventos produzidos por elas. Com isso, os produtores podem manter uma tabela dinâmica com todas as entidades que têm interesse em consumir os eventos que produz. Neste caso, o papel do canal de eventos fica sendo o de atualizar as referências que os produtores possuem a respeito das entidades consumidoras de eventos, para isso, as entidades produtoras de eventos devem registrar quais os tipos de eventos que irão produzir e devem implementar uma interface com um método que permite ao canal de eventos atualizar a relação das entidades consumidoras.

Chamada síncrona direta – Quando as entidades necessitam se comunicar de forma síncrona com chamadas freqüentes a um mesmo serviço, o canal de eventos insere um retardo devido à execução de chamadas a métodos adicionais. Para melhorar a performance nestes casos, as entidades podem

[6] Maiores detalhes podem ser obtidos em [EGS01, EFGK01]

descobrir junto ao canal de eventos qual o objeto servidor e qual o método específico que implementa o serviço requerido. Por conseguinte, após fazer estas chamadas as entidades podem se comunicar diretamente sem a intermediação do canal de eventos. As entidades que fornecem os métodos para comunicação síncrona devem registrar quais os serviços que prestam junto ao canal de eventos previamente.

Eventos compartilhados – O modelo baseado em canais de eventos pode ser utilizado para reger a comunicação entre *threads* de um mesmo processo. Em sistemas onde o número de eventos criados é alto e onde diversas entidades consomem um mesmo tipo de evento, a criação de uma cópia dos eventos recebidos para cada entidade consumidora (*threads*) pode representar um retardo considerável. Nestes casos, ao invés de efetuar várias cópias do evento e repassá-las às entidades consumidoras, pode ser entregue apenas uma referência ao evento original. Diante deste contexto, o acesso aos eventos deve ser limitado a leitura, evitando que as entidades alterem o evento em questão. Assim, quando uma entidade recebe um evento, o contador de referências ao evento é incrementado, ao liberar este evento, após sua utilização, o contador tem seu valor decrescido. Quando não existirem mais referências ao objeto ele é liberado. Esta técnica é viável, pois tipicamente as entidades utilizam os dados de um evento para atualizar seu estado interno ou para produzir novos eventos com alguns dos dados recebidos, não modificando o evento recebido. Assim, as cópias de eventos apenas são realizadas em casos extremos onde uma determinada entidade necessita alterar o evento recebido.

Definição de prioridades – Em alguns casos é interessante que determinados eventos sejam tratados assim que recebidos pelas entidades consumidoras, tendo um tratamento especial em detrimento de outros tipos de eventos. Esta abordagem permite em alguns casos que os eventos subseqüentes sejam tratados mais rapidamente ou que sejam até mesmo descartados. Assim, é comum que os sistemas que implementam o modelo baseado em canais de eventos permitam que sejam atribuídas prioridades para os eventos consumidos [BGH+b00, EFGK01]. As prioridades tipicamente são atribuídas no momento do registro do consumo do evento.

2.4 Conclusão

Apesar do modelo baseado em camadas ter representado um grande avanço no que diz respeito à manutenabilidade e à validação em relação ao modelo plano e de ser na atualidade o mais difundido entre os diversos protocolos de comunicação existentes, ele possui restrições que impactam diretamente na performance dos sistemas:

- **Sobrecarga de tráfego**
- **Ausência de paralelismo**
- **Consumo excessivo de recursos**

Para minimizar os problemas existentes, diversas técnicas de otimização foram propostas. Efetivamente houve ganhos, porém as técnicas, por diversas vezes, quebraram regras do próprio modelo em camadas tornando obscuro o código e improdutivo o trabalho de otimização, pois estas técnicas são aplicadas de forma manual.

Assim, o modelo baseado em canais de eventos se apresenta como uma alternativa para contornar ou ao menos minimizar os principais problemas existentes no modelo em camadas, além de oferecer um paradigma elegante, totalmente orientado a objetos para a comunicação entre processos remotos ou dentro de um mesmo processo.

A principal virtude do modelo baseado em eventos é a comunicação anônima, obtida pela ausência total de acoplamento entre as entidades. Este ponto traz grandes benefícios, maximizando a escalabilidade, flexibilidade, disponibilidade e manutenabilidade do sistema.

Apesar da comunicação anônima ser o ponto mais estudado e divulgado no modelo baseado em canais de eventos [EGS01, EFGK01, OPSS93], há uma série de outras vantagens em relação ao modelo em camadas, em especial no que se refere à resolução de problemas ligados à performance. Podemos observar abaixo que os problemas ressaltados no modelo em camadas são resolvidos de forma natural pelo modelo baseado em canais de eventos:

- **Sobrecarga de tráfego –** Este problema é minimizado, pois apenas são adicionadas as informações de controle das entidades que efetivamente necessitam tratar o evento. Assim, os sistemas pagam pelo que usam. No modelo em camadas, mesmo que uma determinada

camada não necessite tratar um tipo de mensagem, ela o fará agregando informações desnecessárias, pois todas as mensagens devem obrigatoriamente trafegar por todas as camadas da pilha inferiores à camada onde a mensagem foi gerada.

- **Ausência de paralelismo** - O modelo em camadas impõe o tratamento serial das mensagens. Cada camada recebe a mensagem de uma camada adjacente superior ou inferior e a repassa para a próxima camada. Assim, fica claro que é impossível haver paralelismo. No modelo baseado em canais de eventos quando um evento é recebido ele é distribuído para todas as entidades que registraram interesse, assim se *n* entidades estão registradas as *n* receberão o evento e poderão processar em paralelo, obtendo um ganho de performance.
- **Consumo excessivo de recursos** - Apenas as entidades que registram interesse recebem os eventos e os processam. Assim, apenas são consumidos os recursos do sistema estritamente necessários. No modelo baseado em camadas todas as camadas inferiores a que gerou a mensagem necessitam tratar obrigatoriamente a mensagem, mesmo que não tenha nenhuma função naquele caso específico, o que gera um consumo desnecessário dos recursos do sistema.

Portanto, a ênfase deste estudo é fazer uma análise comparativa e experimental, visando verificar na prática as hipóteses levantadas com relação à performance, mensurando o real impacto nos sistemas baseados nos dois modelos estudados.

3 Uma Ferramenta de Comunicação em Grupo Baseada em Camadas

3.1 Introdução

Este capítulo tem como objetivo apresentar a ferramenta de comunicação de grupo (FCG) denominada de iBusTF. O intuito é o de descrever os serviços prestados por ela e fornecer um breve histórico sobre a família de FCGs a que ela pertence.

O iBusTF [CM00] é uma extensão de outra FCG existente, chamada de iBus [MEM99]. Os projetistas do iBusTF criaram uma nova camada, chamada de TF, que adicionou serviços que tornaram o iBus tolerante a falhas. Portanto, o iBusTF é uma versão do iBus **t**olerante a **f**alhas.

A proposta do iBusTF foi a de facilitar a construção de sistemas distribuídos, permitindo que através da replicação ativa [SCH90] estes sistemas mantivessem o funcionamento correto mesmo na presença de falhas de alguns membros do grupo.

Para atender aos requisitos da replicação ativa foi necessário se agregar dois novos serviços ao iBus: (1) ordenação total e (2) visão de grupo atômica. Os protocolos utilizados para implementar estes serviços baseiam-se nos modelos propostos no sistema BCG (Base Confiável de Comunicação em Grupo) [MAC95, GM98], desenvolvido no LaSiD/UFBA (Laboratório de Sistemas Distribuídos da Universidade Federal da Bahia).

Torna-se importante ressaltar a relevância do iBusTF para esta dissertação, pois ele foi a FCG, baseada no modelo em camadas, que serviu de base para a construção do iBusTFE, uma FCG totalmente nova, que provê as mesmas funcionalidades que o iBusTF, porém possui forma de estruturação interna baseada no modelo em eventos. A partir destas duas FCGs foi possível realizar os experimentos e fazer a análise necessária para a execução deste trabalho (vide capítulo 5). Por conseguinte, o entendimento de cada uma das FCGs é fundamental para poder compreender a influencia dos modelos na construção das duas FCGs e, conseqüentemente, nos resultados obtidos nos

experimentos realizados. Maiores detalhes sobre o projeto e a implementação do iBusTF e do iBusTFE podem ser obtidos em [BBa03] e [BBb03], respectivamente.

Na seção 3.2, faremos uma breve descrição sobre o papel das FCGs no desenvolvimento de sistemas distribuídos, serão expostos os serviços prestados pelo iBusTF para as aplicações e definidos alguns termos muito utilizados nesta dissertação. Na seção 3.3, serão apresentadas informações sobre a arquitetura geral do iBusTF e alguns detalhes sobre o projeto e implementação que se aplicam a todas as camadas. Na seção 3.4, todas as camadas serão apresentadas com enfoque especial nos serviços que cada uma presta.

3.2 Serviços de comunicação especializados

A cada dia os sistemas distribuídos têm se tornado mais complexos e abrangentes, novos requisitos como a alta disponibilidade e a tolerância a falhas são demandados com maior freqüência. Por outro lado, para otimizar o desempenho estes sistemas tipicamente utilizam serviços de comunicação básicos não confiáveis, que oferecem um sobrecarga de tráfego menor que os serviços confiáveis.

Assim, os desenvolvedores de sistemas distribuídos convivem com um dilema: obter maior desempenho utilizando protocolos não confiáveis e agregar complexidade à aplicação com o tratamento de perda de mensagens ou pagar o preço (baixo desempenho) pela comunicação confiável. Além disto, se houver a necessidade da comunicação com grupos (um-para-muitos) há novos problemas para as aplicações, como por exemplo, a necessidade de controlar quais membros do grupo estão ativos em determinado momento.

Neste sentido, as FCGs têm fornecido uma grande contribuição aos desenvolvedores de sistemas distribuídos, pois elas encapsulam a complexidade inerente aos serviços de comunicação especializados, oferecendo diversos tipos de serviços como a comunicação *multicast* confiável, a ordenação total de mensagens, o controle de membros ativos etc.

Há diversas FCGs como BCG [MAC95, GM98], Newtop [MES93], Ensemble [MGH98], Horus [RBH94], iBus [MEM99] etc. Cada uma possui características próprias como o modelo de estruturação, a portabilidade, os serviços que prestam às aplicações, entre outras.

Nossos estudos irão se concentrar nos serviços de comunicação providos pelo iBusTF: ordenação total de mensagens, controle de visão atômica do grupo e comunicação *multicast* e *unicast*. Cada um destes serviços será detalhado na seção 3.4 ao apresentarmos as camadas do iBusTF que os provêem.

Ao longo desta dissertação alguns termos serão utilizados com freqüência, com o intuito de expressar alguns conceitos importantes. Abaixo definimos qual o significado específico de cada um destes termos no contexto deste trabalho:

- **Processo ou membro** – É uma aplicação que utiliza uma FCG para se comunicar com um ou mais processos através da troca de mensagens.
- **Grupo** - É um conjunto de processos que tipicamente estão localizados em máquinas distintas e que trocam informações entre si através do envio de mensagens utilizando para isso uma rede de comunicação de dados.
- **Rede de comunicação de dados** – Provê um serviço de entrega de mensagens não confiável. Não oferece garantias com relação ao tempo de entrega de mensagens, sendo considerada desta forma totalmente assíncrona. Falhas podem particionar a rede em subredes e posteriormente as subredes particionadas podem se unir novamente.
- **Falhas dos processos** – Quando ocorrem falhas em um processo consideramos que ele simplesmente irá parar de funcionar e posteriormente será retirado do grupo de processos ativos. Falhas arbitrárias não são consideradas neste trabalho. Para maiores detalhes, favor verificar [LSP82].
- **Visão de grupo** – Ao se criar um grupo é estabelecida uma visão inicial V, que é composta pelos membros ativos naquele instante $V = \{p_1, p_2, p_3 \ldots p_n\}$. Após a instalação da visão inicial, novas visões devem ser geradas quando um processo é incluído ou excluído do grupo. A exclusão de membros pode ser espontânea ou devido a falhas.

3.3 Arquitetura geral

Nesta seção será apresentada a arquitetura geral do iBusTF. Detalharemos como esta FCG interage com as aplicações usuárias, como ela foi subdividida em camadas e como estas camadas interagem entre si para prover os serviços de comunicação previstos.

Os processos podem interagir com o iBusTF para utilizar os seus serviços de três formas distintas: (1) através do envio/recebimento de mensagens para um (*unicast*) ou (2) vários (*multicast*) processos e a (3) notificação de mudanças no grupo. Na Figura 3.1, as operações de envio e recebimento foram denominadas de *send* e *receive,* respectivamente, e as notificações de mudanças no grupo de *view_change*.

Para receber as mensagens enviadas por membros remotos as aplicações devem se registrar previamente junto à FCG fornecendo uma referência para um método de um de seus objetos, que será invocado quando uma mensagem chegar. A mensagem a ser entregue será passada como um parâmetro na invocação desse método. Chamamos este mecanismo de notificação de *callback.*

O serviço de notificação de mudanças nos membros do grupo é utilizado para que os processos sejam informados quando novas visões do grupo são geradas. Estas visões possuem uma lista dos membros ativos e caso a aplicação se registre podem ser enviadas para ela através do mecanismo de *callback*.

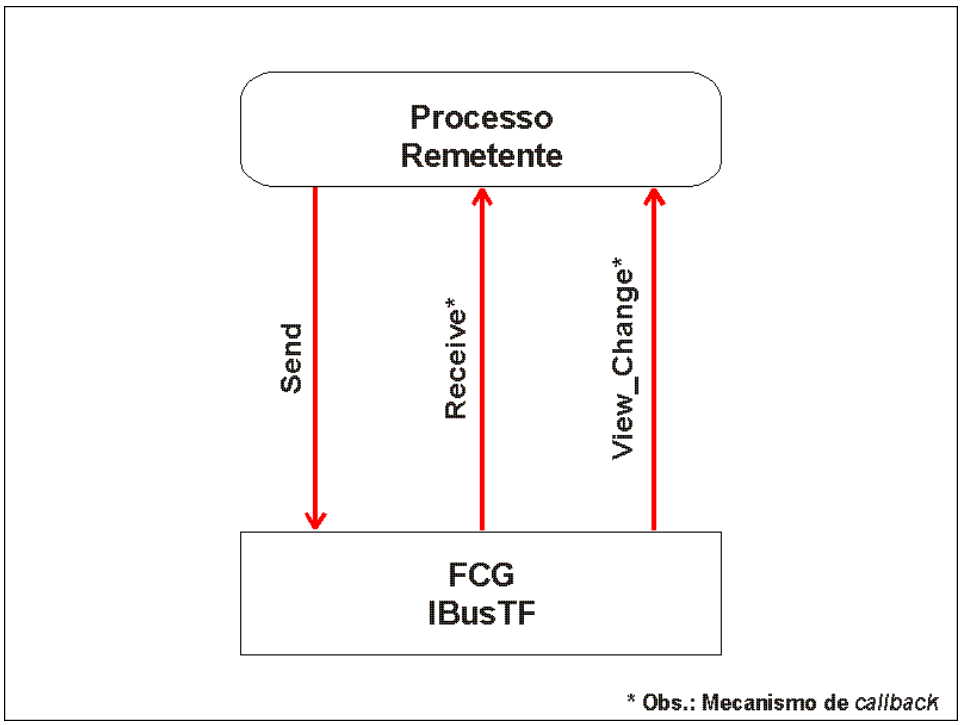

Figura 3.1 – Serviços prestados pelo iBusTF para os processos.

O iBusTF permite que sejam definidos quais os serviços a serem prestados para os processos, através da configuração da pilha de camadas, determinando quais camadas devem compô-la e dessa forma indicando a

qualidade do serviço (QoS – quality of service) requerida. Como o nosso objetivo neste estudo é identificar os impactos que os modelos de estruturação baseado em camadas e em canais de eventos exercem sobre as FCGs estudadas, utilizaremos uma configuração única durante nossos estudos. A implementação do iBusTFE irá espelhar esta mesma configuração, visando manter as funcionalidades estritamente iguais.

Sendo assim, a configuração base utilizada é composta pelas seguintes camadas: STACK, TF-TO, TF-AM, FRAG, FIFO, NAK, REACH e IPMCAST. As camadas foram citadas na ordem que estão dispostas na pilha. A camada STACK é a mais alta na pilha de camadas, a TF-TO é a seguinte e assim em diante. A pilha de camadas do iBusTF pode ser observada na Figura 3.2.

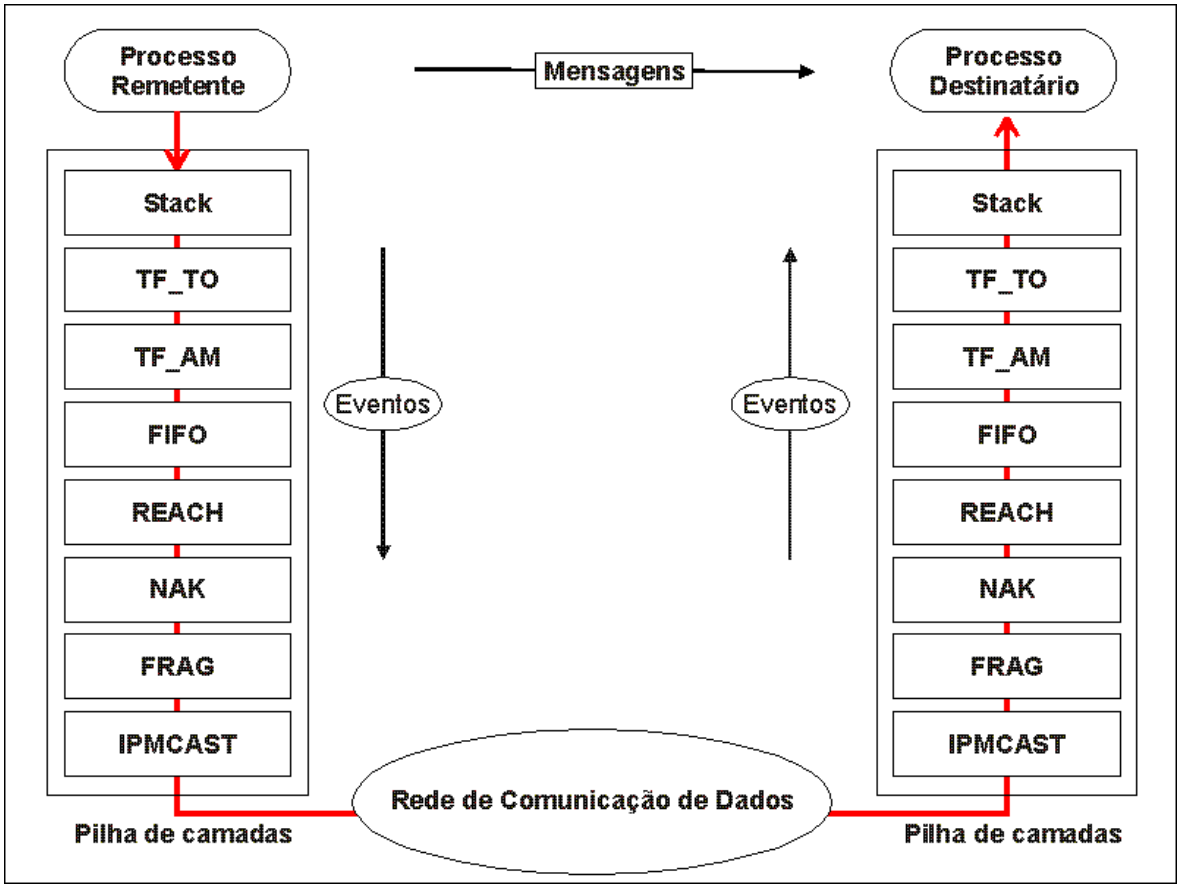

Figura 3.2 - Arquitetura do iBusTF

Esclarecemos que há protocolos de comunicação que disponibilizam serviços de comunicação confiável e que garantem a ordenação FIFO (p.e.: TCP) para a comunicação ponto-a-ponto. Porém, não existe este tipo de serviço para comunicação de um-para-muitos, assim foi necessário criar as camadas FIFO, NAK e FRAG para implementar estes serviços.

Os processos trocam mensagens entre si, estas são encapsuladas em objetos denominados de eventos no iBusTF. Existem diversos tipos de eventos, alguns gerados pelas próprias camadas que serão detalhados no momento oportuno e aqueles gerados a partir de mensagens enviadas pelos processos, denominados *Postings*. Os postings são vetores de objetos serializáveis.

No iBus, foi criada a classe *ProtocolObject* para agrupar as operações e atributos comuns a todas as camadas. Como todas as camadas são descendentes da classe *ProtocolObject* elas herdam os métodos e atributos nela definidos. Este fato está apresentado na Figura 3.3, onde esta representado o diagrama de classe contendo a relação entre as camadas do iBusTF e a classe *ProtocolObject*.

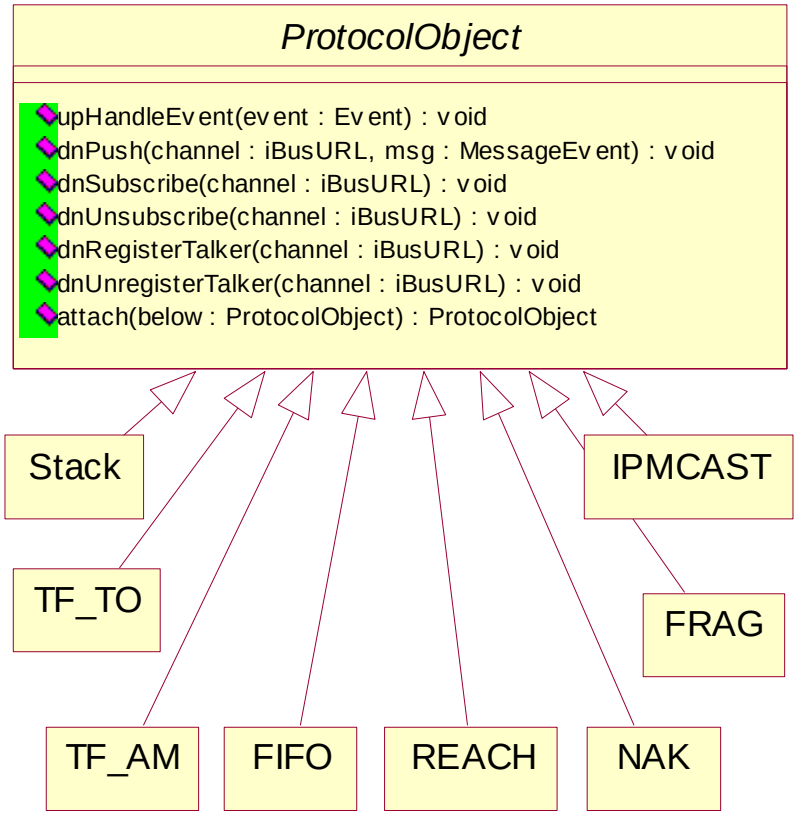

Figura 3.3 – Diagrama de classe do iBusTF contendo a relação entre a classe ProtocolObject e seus descendentes [7].

[7] Todos as figuras que representam diagramas de classes ou de seqüência estão expressos de acordo com a notação definida na linguagem UML.

Na Figura 3.3, também estão descritos os principais métodos definidos pela classe *ProtocolObject*. Quando um método é iniciado pela sigla “dn” indicam que ao ser executado ele repassará informações para a camada adjacente inferior, de forma análoga o método iniciado por “up” repassará informações para a camada adjacente superior.

O último método listado na Figura 3.3 é o **`attach()`**. Ele serve para indicar à camada atual, qual é a camada adjacente inferior e para indicar à camada inferior, qual é a camada superior. Ele é executado por todas as camadas, com exceção da última da pilha, logo após a sua criação quando estiver sendo iniciado. Assim, todas as camadas irão possuir uma referência das camadas adjacentes, o que possibilita o envio de mensagens.

O método **`upHandleEvent()`** é utilizado para entregar um evento à camada imediatamente acima da atual, é passado um objeto do tipo *Event* como parâmetro. O método **`dnPush()`** é utilizado para efetuar a operação contrária, ou seja, para enviar um evento para a camada imediatamente inferior. Neste caso, devem ser repassados dois parâmetros: (1) o endereço destinatário e (2) a própria mensagem a ser enviada. O endereço do destino está encapsulado em um objeto do tipo iBusURL, que contém o endereço IP e a porta do processo destinatário. Caso o endereço IP indicado seja da classe D, esta mensagem será enviada para um grupo de processos (*Multicast)*, caso contrário a mensagem será enviada para um único processo remoto (*Unicast).*

Os métodos **`dnSubscribe()`** e **`dnUnsubscribe()`** servem para que as aplicações se inscrevam ou retirem a inscrição, respectivamente, para receber eventos enviados endereçados a um determinado grupo de processos. Para isso, os processos devem indicar qual o endereço do grupo através do parâmetro do tipo iBusURL.

Os métodos **`dnRegisterTalker()`** e **`dnUnregisterTalker()`** servem para que as aplicações que desejam enviar eventos possam se registrar ou retirar o registro da intenção de envio de eventos.

3.4 Camadas do iBusTF

Nas seções que se seguem serão apresentados os comentários a respeito dos serviços prestados por cada uma das camadas que compõem o iBusTF. Será feita uma abordagem conceitual sobre quais os serviços que cada

camada provê, mantendo a ênfase nas propriedades e garantias fornecidas aos processos.

3.4.1 Camada STACK – Serviços Prestados

Esta camada é a mais alta da pilha de camadas do iBusTF e, conseqüentemente, é a única que interage com os processos que utilizam os serviços de comunicação providos pelo iBusTF. A camada Stack disponibiliza uma interface para acesso aos seguintes serviços: envio e recebimento de mensagens *unicast* e *multicast*, recebimento de visões contendo os processos ativos, configuração da pilha de camadas e possibilidade de incluir e excluir o processo local em grupos de processos para receber e enviar mensagens. A seguir descreveremos cada serviço em detalhes.

3.4.1.1 Criação e configuração de uma instância do iBusTF

A camada Stack permite que os processos criem e configurem a pilha de camadas de acordo com as suas necessidades. Sendo assim, os processos devem indicar quais as camadas que desejam utilizar e devem estabelecer quais e como estão dispostas as camadas na pilha. Cada camada possui parâmetros próprios permitindo que seja feito um ajuste fino da configuração da pilha de camadas, desta forma, os processos estão livres para definirem qual o tipo de serviço que necessitam. Todos os parâmetros necessários para a criação e ajuste fino das camadas serão discutidos ao longo deste capítulo, ao discutirmos cada camada.

3.4.1.2 Inclusão e exclusão em novos grupos de processos

Para que as aplicações enviem ou recebam mensagens de um grupo de processos é necessário que elas se registrem para consumo ou produção de eventos previamente. A camada Stack permite que o processo local faça os registros que necessita, indicando se deseja produzir e/ou consumir mensagens junto a um ou mais grupos. Também é possível retirar um registro, que foi realizado previamente junto a um grupo.

3.4.1.3 Envio e recebimento de mensagens (unicast e multicast)

Essa camada também permite que os processos possam utilizar o serviço de envio de mensagens para um grupo de processos, caracterizando a comunicação *multicast* ou para um único processo (*unicast*).

A qualidade de serviço de entrega de mensagens está diretamente associada à configuração da pilha de camadas. Com a configuração utilizada nos nossos estudos, o serviço relativo ao envio e recebimento de mensagens é confiável e as mensagens são entregues a todos os processos do grupo na mesma ordem (ordenação total). Maiores detalhes sobre a ordenação total e transmissão confiável serão fornecidos ao se abordar as camadas inferiores da pilha, respectivamente as camadas TF-TO e NAK.

3.4.1.4 Recebimento de visões de grupo

Algumas aplicações têm interesse em saber quais membros do grupo estão ativos em um determinado instante, para isso elas registram o seu interesse em receber as visões do grupo que serão geradas. Quando o grupo de processos considerados ativos é alterado, uma nova visão de grupo é gerada, estas alterações ocorrem quando há a inclusão e/ou exclusão de um processo. Um processo pode ser excluído espontaneamente, devido a uma requisição sua ou devido a alguma falha.

É importante frisar que há um mecanismo para a geração e instalação das visões do grupo que envolve todos os membros ativos. Este mecanismo garante que todos os processos ativos integrantes do grupo irão instalar todas as visões do grupo geradas seguindo a mesma ordem, chamamos esta propriedade de visão de grupo atômica. Maiores detalhes serão fornecidos ao abordarmos a camada TF-AM, responsável por disponibilizar este serviço.

3.4.2 Camada TF-TO – Serviços Prestados

3.4.2.1 Ordenação total de mensagens

A camada TF-TO tem como principal objetivo garantir que todas as mensagens enviadas a um grupo sejam entregues de forma totalmente ordenada. Todos os membros do grupo deverão receber o mesmo conjunto de mensagens e todas na mesma ordem, assim, a relação de causalidade existente entre as

mensagens é preservada. Maiores detalhes sobre a definição de ordenação total utilizada neste trabalho, podem ser encontrados em [MAC95].

Para garantir a ordenação total, a camada TF-TO se baseia nos serviços prestados pelas camadas inferiores[8], relacionados a seguir:

- As mensagens enviadas ao grupo devem ser recebidas ordenadas seqüencialmente, com base no número de seqüência gerado pela camada NAK. Esta funcionalidade é provida pela camada FIFO.
- Deve haver a garantia que todas as mensagens enviadas ao grupo serão entregues a todos os membros ativos. Portanto, é necessário haver um mecanismo para a recuperação de mensagens perdidas. As camadas NAK e TF-AM tratam este problema.
- É necessário que todos os membros do grupo possuam a mesma visão do grupo instalada (funcionalidade provida pela camada TF-AM).

Portanto, a camada TF-TO parte do pressuposto que a qualidade do serviço de comunicação que necessita será garantida pelas camadas inferiores. Assim, esta camada pode implementar o mecanismo para ordenação das mensagens. Para isso, ela utiliza uma abstração central chamada de matriz de blocos. Esta matriz tem o objetivo de registrar todas as mensagens recebidas do grupo. Cada processo integrante do grupo deve possuir uma matriz própria.

As colunas da matriz de blocos representam os membros do grupo e as linhas os blocos causais. Um bloco causal é um vetor numerado seqüencialmente que possui o tamanho do grupo em questão. Todas as mensagens possuem dois atributos fundamentais para a camada TF-TO: (1) um número que identifica a que bloco causal elas pertencem e (2) a identificação de qual foi o processo que a remeteu para o grupo. Com estas informações (processo X bloco causal) é possível localizar e registrar na “célula” apropriada da matriz de blocos o recebimento da mensagem em questão.

O número que associa as mensagens aos blocos causais é definido através da utilização de um esquema de sincronização, implementado pela camada TF-TO, que envolve todos os processos integrantes do grupo. O

[8] As funcionalidades aqui citadas serão detalhadas neste capítulo, em seções posteriores, ao comentarmos cada camada.

esquema de sincronização funciona da seguinte maneira: cada membro ao ser iniciado cria um contador interno com o valor “0” que representa o valor do bloco causal local. Este contador pode ser incrementado de duas formas distintas: (1) a cada mensagem enviada é adicionada uma unidade ao contador ou (2) ao receber uma mensagem do grupo que possua um valor do bloco causal maior que o contador local, então é atribuído este valor ao contador local[9].

Assim, podemos garantir que um membro do grupo após enviar uma mensagem referente ao bloco causal de número X jamais irá gerar posteriormente outra mensagem com bloco causal Y, onde Y < X.

Baseado nesta garantia, podemos inserir outro conceito importante: o de bloco causal completo. Um bloco causal atinge a condição de completo quando ele está relacionado a um bloco causal X e em determinado momento é possível concluir que nenhum integrante do grupo poderá enviar uma mensagem com identificador de bloco causal menor que X. Podemos chegar a esta conclusão quando todos os membros já tiverem enviado mensagens com número de bloco maior ou igual a X.

Portanto, todos os blocos causais com identificador menor ou igual a X são considerados completos. Apenas as mensagens relacionadas a blocos causais completos podem ser entregues à camada Stack. Antes de enviar as mensagens à camada superior, elas devem ser ordenadas utilizando um algoritmo determinístico. Desta forma, pode-se garantir que todos os integrantes de um grupo receberão as mensagens na mesma ordem.

Para exemplificar os conceitos apresentados, mostramos na Tabela 3.1 a matriz de blocos do processo P1. Observando as colunas existentes na Tabela 3.1 podemos concluir que o grupo em questão possui quatro integrantes: P1, P2, P3 e P4 e que até o momento foram emitidas mensagens relacionadas a 5 blocos causais que são representados pelas linhas da matriz.

	P1	P2	P3	P4
Bloco Causal 1	■			
Bloco Causal 2		■	■	

[9] O esquema de sincronização entre processos distribuídos citado foi descrito por Lamport [LAM78] e implementa uma espécie de relógio lógico entre os vários processos.

Bloco Causal 3		■		■
Bloco Causal 4	■			
Bloco Causal 5			■	

Tabela 3.1 – Exemplo de matriz de blocos

Podemos verificar que as últimas mensagens recebidas de cada membro possuem os seguintes números de blocos causais: 4, 3, 5 e 3, respectivamente. Logo, podemos concluir que nenhum integrante do grupo poderá enviar mensagens com bloco causal igual ou inferior a 3, devido ao esquema de sincronização citado anteriormente. Assim, os blocos 1, 2 e 3 podem ser considerados completos e as mensagens relativas a eles podem ser entregues para a camada Stack.

Se um membro permanecer por um longo período sem gerar novas mensagens os blocos causais não poderão ser considerados completos e conseqüentemente as mensagens não serão entregues para a camada Stack. Para contornar este problema é necessário que os processos enviem mensagens nulas, caso o processo local fique inativo por um tempo determinado. O intervalo de tempo para iniciar o envio de mensagens nulas é parametrizável e deve ser indicado quando a camada TF-TO é criada.

Os blocos causais devem ser removidos da matriz de blocos após existir a garantia de que todos os integrantes do grupo receberam as mensagens. Para isso, os integrantes enviam no cabeçalho das mensagens a informação sobre qual é o último bloco completo. Podem ser removidos da matriz todos os blocos com identificador igual ou inferior ao menor bloco completo informado pelos integrantes do grupo.

A camada TF-TO foi implementada através da construção de uma classe chamada de *Newtop*. Esta classe foi reutilizada, para implementar a ordenação total, pelo iBusTFE.

3.4.3 Camada TF-AM – Serviços Prestados

3.4.3.1 Visão de grupo atômica

Uma visão de grupo é um retrato deste em um determinado instante. Ela contém a relação de todos os membros ativos que estão inscritos no grupo. A

primeira visão do grupo é estabelecida quando um grupo é criado, ela é composta pelos membros ativos naquele instante. Após a instalação da visão inicial, são geradas novas visões quando há uma inclusão e/ou exclusão de um processo no grupo. A exclusão de um membro do grupo pode ser espontânea ou motivada por uma falha em um processo.

A camada TF-AM é responsável por garantir que todos os membros ativos do grupo convirjam para ter uma única visão instalada, ou seja, que todos tenham a mesma visão sobre quais os processos que integram o grupo em um dado instante. Para isso, ela utiliza o serviço de visões de grupo prestado pela camada REACH. Este serviço é implementado através do envio de mensagens de *HeartBeat.* Todos os processos devem transmitir para o grupo periodicamente, em um intervalo de tempo X, mensagens chamadas de *HeartBeats*. A função desta mensagem é indicar que o remetente está ativo. Assim, os membros recebem periodicamente *HeartBeats* de todos os outros integrantes do grupo. Caso algum membro não receba o *HeartBeat* de outro integrante, em um intervalo de tempo Y, onde Y >= X, então é gerada uma nova visão do grupo localmente, excluindo o membro cuja mensagem não foi recebida.

Porém, é importante atentar para o fato de que a rede de comunicação utilizada é assíncrona e provê serviços de comunicação não confiáveis. Portanto, as mensagens de *HeartBeat* podem ser perdidas ou mesmo serem entregues com atraso. Nestes casos podem ser geradas novas visões locais excluindo membros que na realidade não falharam, fica claro que as visões do grupo geradas pela camada REACH podem não espelhar a realidade, pois existe a possibilidade de excluírem membros que ainda estão ativos. Por isso, definimos as visões geradas pela camada REACH como “temporárias”[10].

Outra limitação do serviço de visões provido pela camada REACH é o fato de que podem existir diferentes visões do grupo instaladas nos membros ativos que integram o grupo em um dado instante. Isto ocorre, pois existe a possibilidade de perda de mensagens na rede de comunicação provocando a geração de visões de grupo falsas em um ou mais membros do grupo.

Cabe à camada TF-AM garantir que todos os membros considerados ativos instalem as diferentes visões do grupo geradas na mesma ordem,

[10] Maiores detalhes sobre a camada REACH e seus serviços no decorrer deste capítulo.

convergindo para que todas possuam a mesma versão instalada, chamamos esta propriedade de visão de grupo atômica. Para implementá-la existe um mecanismo para a exclusão de um membro do grupo que promove um consenso entre os membros ativos, antes de instalar uma nova visão que exclua um membro.

Apesar de garantir que todos convirjam para ter a mesma visão instalada, a camada TF-AM não pode garantir que membros ativos não sejam excluídos erroneamente. Isto ocorre porque em ambientes onde a comunicação é assíncrona, não é possível detectar através de um consenso determinístico que um membro falhou, apenas é possível gerar uma suspeita de falha, se for inserido um tempo de espera padrão. Neste tipo de ambiente não há qualquer garantia quanto ao tempo de resposta, assim é impossível determinar se um membro não enviou uma mensagem ou se ela ainda está a caminho [POW96].

O mecanismo que a camada TF-AM utiliza para exclusão de um membro é disparado quando uma visão “temporária” da camada REACH é recebida excluindo algum membro. Ao receber esta nova visão a camada TF-AM gera uma mensagem chamada *Suspect* e a envia para todos os membros considerados ativos naquele instante, esta mensagem contém o número identificador da última mensagem recebida do membro suspeito de ter falhado.

Os integrantes do grupo ao receberem uma mensagem *Suspect,* verificam qual foi a última mensagem recebida localmente e comparam seu número identificador com o número que está indicado na mensagem de suspeita. Desta forma, é possível determinar se o membro local recebeu mensagens posteriores à suspeita da falha. Neste caso, os membros irão adotar uma das duas alternativas abaixo:

- Caso o membro local tenha recebido alguma mensagem com o número identificador maior que o indicado na mensagem de suspeita, ele deverá rejeitar a suspeita enviando uma mensagem *Refute* que indica qual a última mensagem recebida.
- Se o membro local não recebeu mensagens posteriores à falha, ele não deverá tomar nenhuma atitude. Em breve ele mesmo irá gerar uma suspeita da falha e enviar um evento do tipo *Suspect* para o grupo.

Assim que todos os membros considerados ativos enviarem mensagens *Suspect* para o grupo é gerada uma nova visão.

Portanto, quando é gerada uma mensagem de suspeita podemos ter duas situações distintas: (1) o membro realmente falhou ou (2) a suspeita gerada era falsa e o membro suspeito continua ativo.

Quando o membro falhou, em determinado momento, todos os integrantes do grupo irão suspeitar da falha e gerar mensagens de suspeita. Assim que o primeiro membro que suspeitou da falha obtiver as mensagens confirmando a suspeita de todos os membros ativos, ele irá gerar uma mensagem do tipo *Confirmed,* confirmando a falha do membro. Quando um processo ativo recebe uma mensagem confirmando a falha de algum membro, ele irá excluir o membro que falhou gerando uma nova visão do grupo.

No segundo caso, quando a suspeita de falha é falsa, se ao menos um membro do grupo tiver recebido mensagens posteriores à suspeita da falha, ele irá verificar o recebimento das mensagens posteriores à suspeita e irá refutá-la enviando ao grupo uma mensagem do tipo *Refute* contendo o número da última mensagem recebida.

Quando uma suspeita é refutada por algum membro, significa dizer que o membro que gerou a suspeita necessita recuperar as mensagens que não recebeu. Para isso, ele deve solicitar ao grupo a recuperação das mensagens perdidas, através do envio de uma mensagem do tipo *Recovery* indicando qual a última mensagem que foi recebida. Os membros do grupo deverão atender à requisição de recuperação de mensagens enviando todas as mensagens recebidas após a mensagem indicada na mensagem *Recovery*.

Para ilustrar o mecanismo utilizado pela camada TF-AM, citamos a seguir dois exemplos distintos onde são geradas suspeitas de falhas de um membro. No primeiro exemplo a suspeita é confirmada (Figura 3.4) e no segundo é refutada (Figura 3.5).

Na Figura 3.4 o grupo em questão é composto originalmente por quatro membros: P1, P2, P3 e P4. Em um determinado instante o membro P4 falha, então deverá ser iniciado o processo de geração de uma nova visão. O primeiro membro a suspeitar da falha é P1, a camada TF-AM intercepta uma nova visão enviada pela camada REACH e gera um evento *Suspect* para o grupo. Assim, P1 está indicando para o grupo que há uma suspeita de falha em relação a P4 e que a última mensagem recebida dele tem número de bloco "8". Os outros membros ativos (P2 e P3) recebem a suspeita gerada por P1 e não a refutam. Logo em

seguida P2 e P3 geram suas próprias suspeitas a respeito da falha de P4 e enviam os eventos *Suspect* para o grupo. Quando P1 recebe o evento de suspeita gerado por P2 ele não estabelece o consenso para exclusão do membro suspeito, pois falta a suspeita de P3. Porém, ao receber a última suspeita, gerada por P3, o consenso é estabelecido e P1 envia o evento *Confirmed* excluindo P4 e instala uma nova visão do grupo localmente. Assim, que recebem o evento *Confirmed,* P2 e P3 instalam a nova visão do grupo composta por P1, P2 e P3.

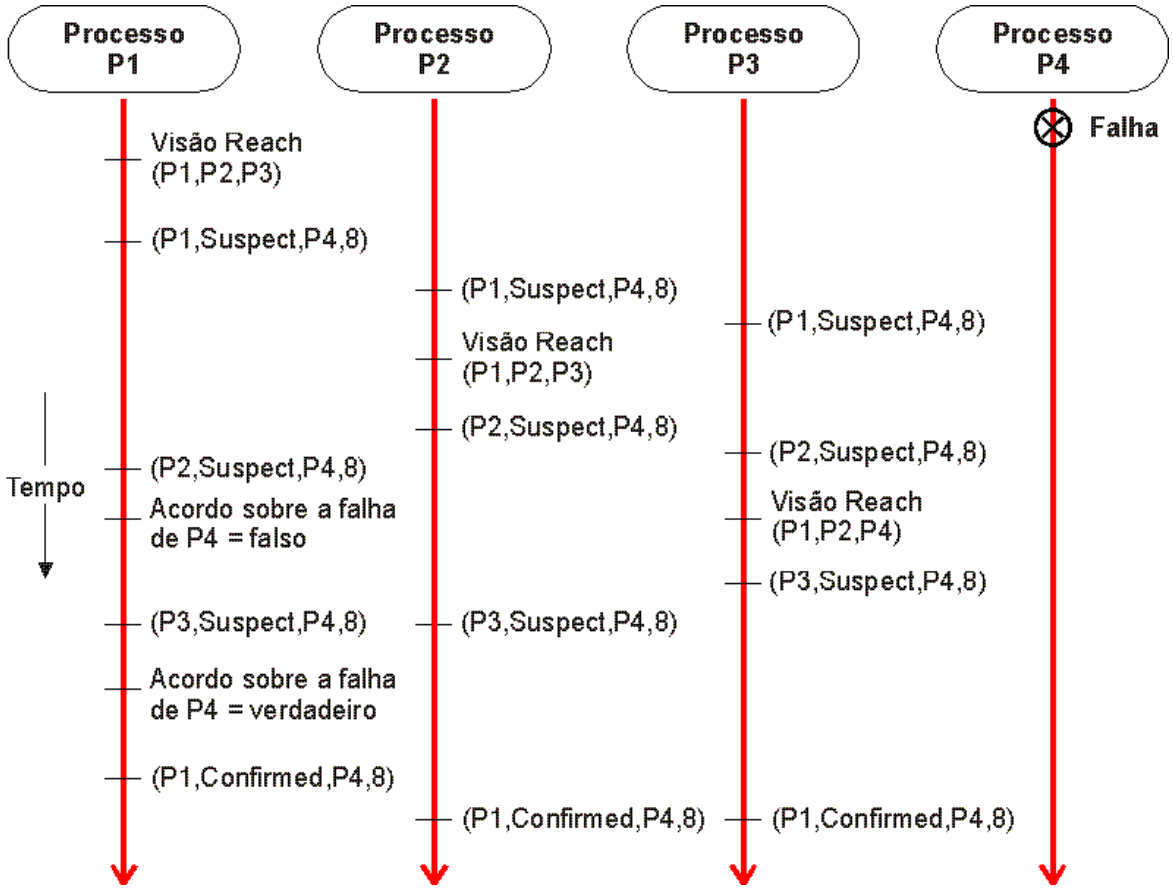

Figura 3.4 - Acordo para confirmar a suspeita de falha de um membro.

Na Figura 3.5 podemos observar um acordo para exclusão de um membro suspeito que foi refutado, pois a suspeita de falha era falsa. Neste exemplo, a camada REACH do membro P1 gera uma visão excluindo o P4 de forma errônea. Ao receber a nova visão a camada TF-AM gera um evento *Suspect* indicando que a última mensagem recebida foi a de número 8. O membro P4 recebe a mensagem de suspeita e não toma nenhuma atitude, apenas aguarda que os outros membros refutem a suspeita. Quando P2 e P3 recebem a suspeita, verificam que a última mensagem que receberam de P4 foi a mensagem "9" e não a “8” como indicado na mensagem *Suspect*. Portanto, eles geram

mensagens refutando a suspeita. Assim que P1 recebe as mensagens *Refute*, ele abandona o consenso para exclusão do membro e inicia o processo de recuperação das mensagens que não foram recebidas. Para isso P1 envia uma mensagem *Recovery* solicitando todas as mensagens com número igual ou superior a "9". Então, os membros ativos do grupo enviam a mensagem “9” para P1, concluindo o processo de recuperação de mensagens.

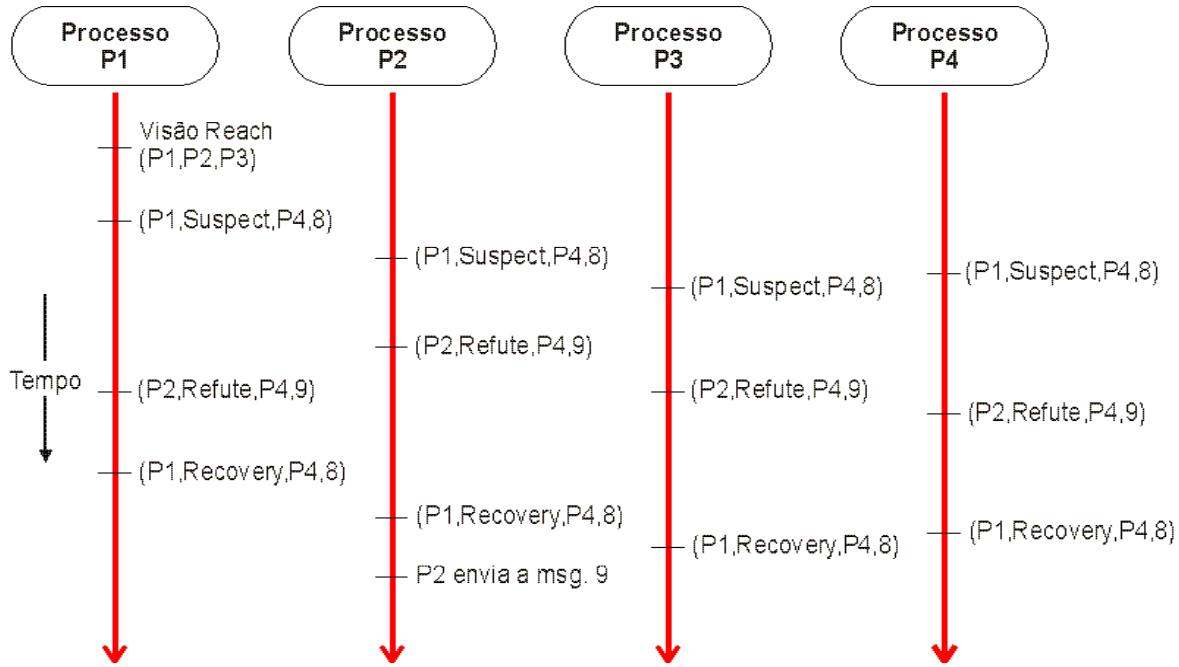

Figura 3.5 - Acordo para refutar a suspeita de falha de um membro

3.4.4 Camada FRAG – Serviços Prestados

3.4.4.1 Fragmentação e reconstrução de mensagens

Esta camada tem o propósito de fragmentar, no processo remetente, as mensagens enviadas pela aplicação que excedam a um tamanho pré-definido e reconstruir a mensagem original, no processo destinatário, com base nos fragmentos de mensagem recebidos.

Desta forma, a aplicação pode gerar mensagens sem qualquer restrição quanto ao tamanho, pois a partir dela podem ser produzidos um ou vários pacotes. Estes pacotes possuem números que os identificam permitindo que sejam unidos no seu destino, formando a mensagem original. O tamanho padrão do pacote é equivalente ao tamanho de um datagrama UDP (8192 bytes), porém este valor pode ser alterado quando a camada FRAG é criada, basta para isso indicar o tamanho desejado como parâmetro de inicialização (vide seção 3.4.1.1).

Apenas os eventos gerados pela aplicação, ou seja, os do tipo *MessageEvent,* são fragmentados e reconstruídos*.* Os eventos que são gerados pelas camadas superiores (Stack TF-TO e TF-AM) não são tratados, eles são enviados aos destinatários como foram gerados. A camada FRAG pode consumir e/ou produzir eventos do tipo *MessageEvent* em dois momentos distintos:

- Quando recebe uma mensagem gerada pela aplicação e a repassa para baixo na pilha de camadas. Neste caso, é consumida uma e produzida uma ou mais mensagens do tipo *MessageEvent.* A mensagem gerada pela aplicação local é recebida da camada TF-AM e a partir dela são produzidos um ou vários fragmentos que são remetidos para o grupo. Esses fragmentos são objetos do tipo *MessageEvent* que possuem tamanho menor ou igual ao pré-definido na inicialização da camada FRAG.
- Quando recebe uma ou várias mensagens da camada FIFO que devem ser reagrupadas e enviadas para cima na pilha de camadas. Podem ser consumidos diversos eventos do tipo *MessageEvent*, enviados por um processo remoto, que representam apenas uma única mensagem a ser entregue para a aplicação. Então é produzido um único evento que representa a mensagem originalmente criada pela aplicação remetente e que é enviada para a camada imediatamente superior (TF-AM).

A camada FRAG assume que não há a perda de fragmentos de mensagens e que eles serão recebidos na mesma ordem em que foram enviados. Para garantir que estes dois requerimentos sejam verdadeiros é necessário que a pilha de camadas possua abaixo da camada FRAG as camadas NAK e FIFO, que são responsáveis por garantir a qualidade de serviço requerida pela camada FRAG.

3.4.5 Camada FIFO – Serviços Prestados

O serviço prestado por esta camada apenas é utilizado no recebimento de mensagens. Quando uma mensagem é gerada localmente, para ser enviada ao grupo, a camada FIFO apenas repassa essa mensagem sem nenhum tratamento especial. A seguir abordaremos os serviços prestados pela camada FIFO.

3.4.5.1 Ordenação seqüencial de mensagens

A camada FIFO garante que todas as mensagens que forem recebidas do grupo serão enviadas à camada FRAG ordenadas seqüencialmente. Para isso, é utilizado um número de seqüência atribuído pela camada NAK que está contido em todas as mensagens do tipo *MessageEvent*.

As informações a respeito dos processos inscritos como produtores de mensagens e que estão ligados aos grupos em que o processo local está inscrito para recebimento de mensagens são as seguintes:

- A identificação do remetente;
- A identificação do grupo;
- Qual o número da última mensagem enviada para a camada FRAG enviada pelo membro em questão;
- Todas as mensagens recebidas fora da ordem

Portanto, quando uma mensagem é recebida são localizadas as informações a respeito do remetente da mensagem e então é realizada uma análise para verificar se a mensagem foi recebida de forma ordenada. Em caso afirmativo, ela é enviada para cima na pilha de camadas, caso contrário é armazenada até que as mensagens necessárias para garantir a ordenação sejam recebidas, quando então ela será enviada para a camada superior na ordem correta.

As informações a respeito dos processos do grupo que produzem mensagens são obtidas a partir de eventos do tipo *View,* gerados pela camada REACH. Quando a camada FIFO recebe um evento deste tipo ela verifica se houve alguma alteração no grupo e caso seja necessário, ajusta a tabela de processos que produzem mensagens para espelhar a nova formação do grupo.

Para funcionar corretamente a camada FIFO assume que não haverá perda de mensagens, pois ela não implementa nenhum tipo de mecanismo de recuperação de mensagens. Para isso, é necessário que a camada NAK seja configurada abaixo da camada FIFO na pilha de camadas.

3.4.5.2 Remoção de mensagens duplicadas

Outra garantia provida pela camada FIFO é a de que não serão entregues mensagens em duplicidade para a camada FRAG. Para evitar este fato, sempre que uma mensagem é recebida, as informações a respeito do processo remetente

são recuperadas e é feita uma comparação entre a última mensagem entregue de forma ordenada à camada FRAG e a mensagem recebida no momento. Caso o número seqüencial da mensagem em questão seja menor ou igual ao número da última mensagem entregue, isto significa que a mensagem em questão foi entregue anteriormente, sendo assim, ela deverá ser descartada.

3.4.6 Camada NAK – Serviços Prestados

3.4.6.1 Recuperação de Mensagens Perdidas

A camada NAK implementa o serviço de recuperação de mensagens que torna o iBusTF confiável. O mecanismo utilizado é baseado no envio de mensagens de confirmação por parte dos processos consumidores das mensagens, ou seja, cada membro do grupo inscrito como consumidor envia uma mensagem para confirmar o recebimento de um grupo de mensagens, aqui chamado de época. Para cada época recebida de um determinado processo remoto, deve ser enviada uma mensagem chamada de Ack (do inglês **ack**nowledgment) confirmando o recebimento.

Ao criar a camada NAK é necessário definir qual o tamanho da época a ser utilizada por todos os membros do grupo, ou seja, quantas mensagens deverão ser recebidas antes que seja gerado um evento confirmando o recebimento delas. As mensagens possuem um número seqüencial crescente (numSeq), iniciado com valor zero, que possibilita aos processos que as recebem identificar se houve perda de mensagens e a que época elas pertencem. Por exemplo, digamos que um grupo tenha definido o tamanho da época em 100 mensagens, assim se for recebida a mensagem com numSeq equivalente a 59 é possível determinar que ela faz parte do época 1, se for recebida outra mensagem com numSeq 103 ela pertence à época 2 e assim por diante.

As mensagens enviadas ao grupo são armazenadas pelo membro remetente até que este tenha a confirmação do recebimento por parte de todos os membros ativos do grupo. Assim, caso haja alguma mensagem perdida ela poderá ser retransmitida pelo membro que originalmente a remeteu.

Os membros que recebem as mensagens registram que elas foram recebidas com sucesso e as repassam para as camadas superiores. Desta forma, quando a última mensagem de uma época for recebida será possível determinar

se houve a perda de alguma mensagem ou se deve ser enviada uma mensagem Ack imediatamente. Isso somente é possível, pois o numSeq é um número inteiro, seqüencial e crescente. Sendo assim, não pode haver intervalos entre o numSeq de duas mensagens consecutivas, se houver é sinal de que uma ou mais mensagens foram perdidas.

Ao receber a primeira mensagem de uma nova época as estruturas que armazenam o recebimento da época anterior são inicializadas, apagando-se as referências das mensagens recebidas, e passam a trabalhar com a nova época.

Sempre que recebem uma mensagem, os membros do grupo verificam se a época atual está completa, em caso afirmativo é enviado um Ack. Isto garante que será enviado de imediato um Ack quando forem recebidas mensagens que completam a época atual e que estão fora da ordem esperada ou mesmo que foram retransmitidas pelo membro remoto.

Será suspenso o envio de mensagens, enquanto o membro que remeteu uma época completa não receber uma mensagem Ack de cada membro ativo do grupo. Ao recebê-las, ele reiniciará o envio das mensagens, criando uma nova época, descartando as mensagens da época anterior, registradas previamente. Isto é possível, pois nesse momento há a certeza de que todos os membros do grupo receberam a época em questão com sucesso. A exemplo das mensagens, as épocas possuem um identificador que é um número inteiro, seqüencial e crescente.

Para auxiliar na apresentação do mecanismo de recuperação de mensagens partiremos do pressuposto que existe um grupo P, composto pelos membros $\{P_1, P_2, P_3\}$ que trocam mensagens entre si $\{M_1,M_2,\ldots,M_n\}$ e que o tamanho da época definido para utilização pelos membros do grupo P é de 10 mensagens.

Na Figura 3.6 podemos observar o mecanismo utilizado pela camada NAK para confirmar o recebimento de uma época. No exemplo exposto, estão representados os processos que integram o grupo P. O membro P1 envia as mensagens de M0 a M8 ao grupo, utilizando o método **`dnPush()`**. Os membros P2 e P3 recebem as mensagens através do método **`upHandleEvent()`** e as processam, registrando o recebimento e verificando se elas completam a época a que pertencem. Como nenhuma delas completa a época, então P2 e P3 simplesmente aguardam novas mensagens. Quando P1 envia a mensagem M9,

última mensagem da época, ele bloqueia o envio de novas mensagens até que todos os Acks sejam recebidos. Os membros P2 e P3 recebem e processam M9, registrando o seu recebimento. Constatam que todas as mensagens da época 1 foram recebidas com sucesso. Assim, P2 e P3 enviam uma mensagem de Ack cada. Quando P1 recebe os Acks de todos os membros considerados ativos do grupo (P2 e P3), reinicia o envio de mensagens remetendo a mensagem M10, que já pertence a uma nova época.

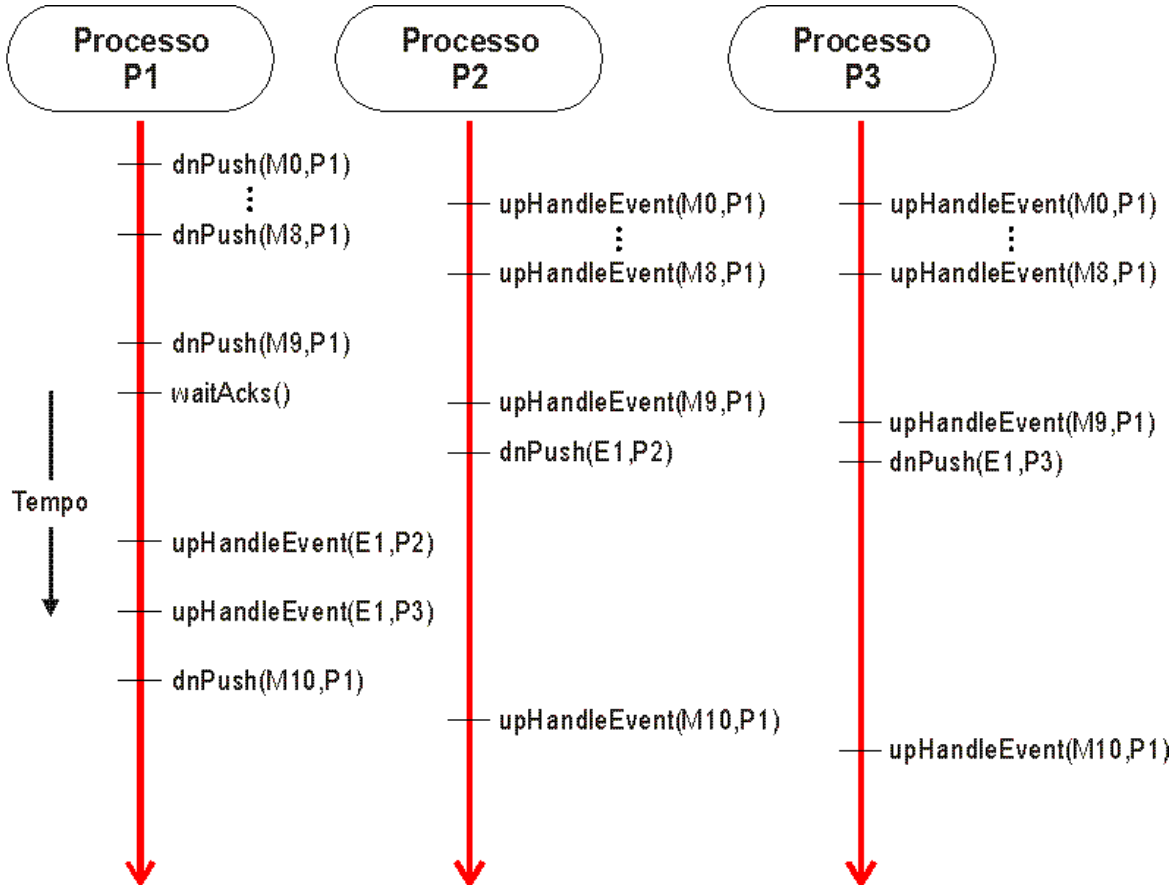

Figura 3.6 - Envio de mensagens de confirmação de recebimento de uma época.

Apresentamos no parágrafo anterior, um exemplo que mostra como funciona a camada NAK quando não existe perda de mensagens. A seguir, iremos discutir como ela se comporta na presença de falhas, ou seja, em uma situação onde é necessário haver a retransmissão de mensagens devido a algum problema que ocasiona o não recebimento de uma ou mais mensagens por parte de um ou mais membros do grupo.

Consideramos que houve uma falha no envio/recebimento de uma mensagem quando um membro remetente (P0) envia uma mensagem M, com

destino a um grupo de processos P e esta mensagem não é recebida por P1, onde este representa um processo que faz parte de P. Neste caso, o mecanismo para recuperação de mensagens deve ser acionado para que a mensagem M seja retransmitida por P0, permitindo que P1 receba novamente a mensagem perdida.

O mecanismo de retransmissão de mensagens é acionado assim que uma mensagem com o numSeq maior que o da mensagem perdida for recebida, ou seja, caso alguma mensagem não seja recebida na ordem correta. Este fato pode ser identificado, pois o numSeq é um número inteiro seqüencial crescente, sendo assim, não pode haver intervalos entre mensagens. Portanto, se recebermos uma mensagem com o numSeq igual a 3, podemos deduzir que a próxima mensagem a ser recebida deve ter o numSeq equivalente a 4. Se recebermos alguma mensagem com numSeq maior que 4 então o mecanismo de recuperação de mensagens deverá ser acionado. Neste caso, a camada NAK presume que houve uma falha no recebimento das mensagens e solicita ao membro remetente que a mensagem seja enviada novamente ao grupo.

Esta solicitação é efetuada através do envio de uma mensagem do tipo *RetransmitReq*, que poderá solicitar o reenvio de uma ou mais mensagens. O único membro que irá tratá-la será o remetente original da mensagem perdida. Ao recebê-la ele irá buscar todas as mensagens solicitadas no pedido de retransmissão e encaminhá-las novamente ao grupo. A retransmissão de mensagens pode provocar a duplicação e a entrega fora da ordem de envio de mensagens. A camada NAK não trata estes problemas, estes serviços são oferecidos por camadas superiores (FIFO e TF-TO).

Quando ocorre uma falha no recebimento da última mensagem de uma época, é necessário haver um tratamento especial. Nestes casos, o membro que não recebeu a mensagem devido à falha, não confirma o recebimento da época completa e fica aguardando receber a última mensagem. Paralelamente, o remetente da época em questão suspende o envio de novas mensagens, pois está aguardando receber os Acks do grupo. Este fato pode provocar a suspensão eterna do envio de mensagens por parte do membro remetente. Para contornar este “travamento” do sistema, são enviados eventos especiais chamados de *HeartBeat* indicando qual a última mensagem enviada pelo seu remetente. Essas mensagens são remetidas quando o membro local fica um período de tempo pré-

determinado sem enviar nenhuma mensagem. A partir deste momento são enviados *HeartBeats* de forma periódica[11]. O "travamento" relatado é evitado, pois quando o membro que perdeu a última mensagem da época receber o *HeartBeat* ele irá verificar que falta receber uma ou mais mensagens e solicitará a retransmissão das mensagens através do envio de uma mensagem *RetransmitReq*.

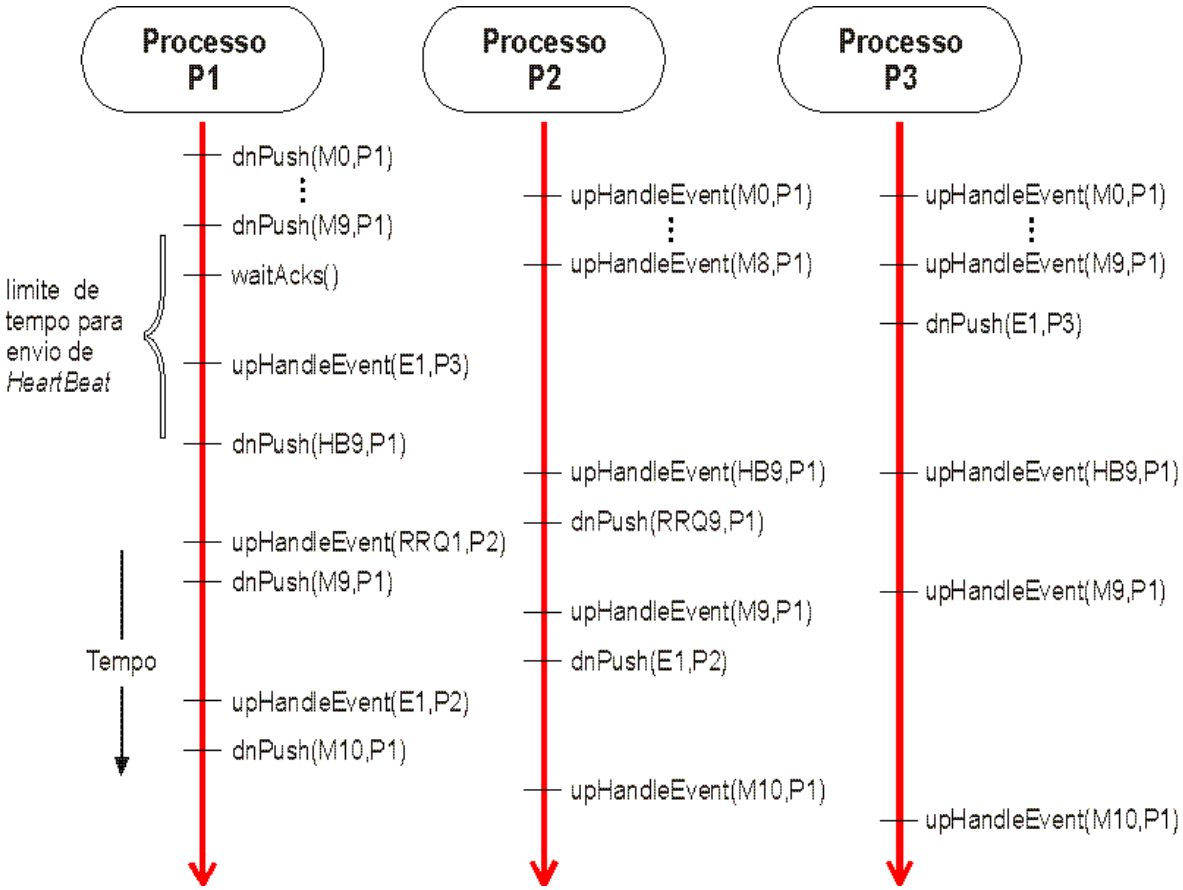

Figura 3.7 - Recuperação de mensagens com a utilização de HeartBeats

Na Figura 3.7 apresentamos como o mecanismo de recuperação de mensagens funciona quando a última mensagem da época é perdida. Neste exemplo, o membro P1 envia as dez mensagens pertencentes à primeira época (M0 a M9). O membro P2 recebe todas, exceto a mensagem M9, e P3 recebe todas as mensagens com sucesso. Assim, P2 deverá solicitar a retransmissão da mensagem que não recebeu.

[11] Os intervalos de tempo para iniciar o envio dos *HeartBeats* e para determinar qual o periódo para enviar novos eventos devem ser definidos no momento de criação da camada FRAG.

O membro P2 somente terá conhecimento da falha após receber o primeiro *HeartBeat* enviado por P1. Note que o *HeartBeat* somente é enviado após um período de tempo em que P1 não envia mensagens, aguardando o recebimento dos Acks. Ao receber o *HeartBeat,* P2 constata que M9 já foi remetido por P1, porém não foi recebido localmente indicando a necessidade de retransmissão. Logo, será gerado um evento *RetransmitReq* pelo membro P2 indicando a necessidade de retransmissão de M9. Quando P1 recebe a mensagem enviada por P2, solicitando a retransmissão, então ele recupera e envia novamente ao grupo a mensagem M9. Desta forma, P2 e P3 recebem M9. Este fato resolve o problema de P2 que recebe M9 e envia um Ack, porém gera uma duplicidade no recebimento da mensagem M9 por parte de P3. Portanto, nestes casos a camada NAK envia por duas vezes a mensagem M9 para a camada FIFO que deverá tratar este problema.

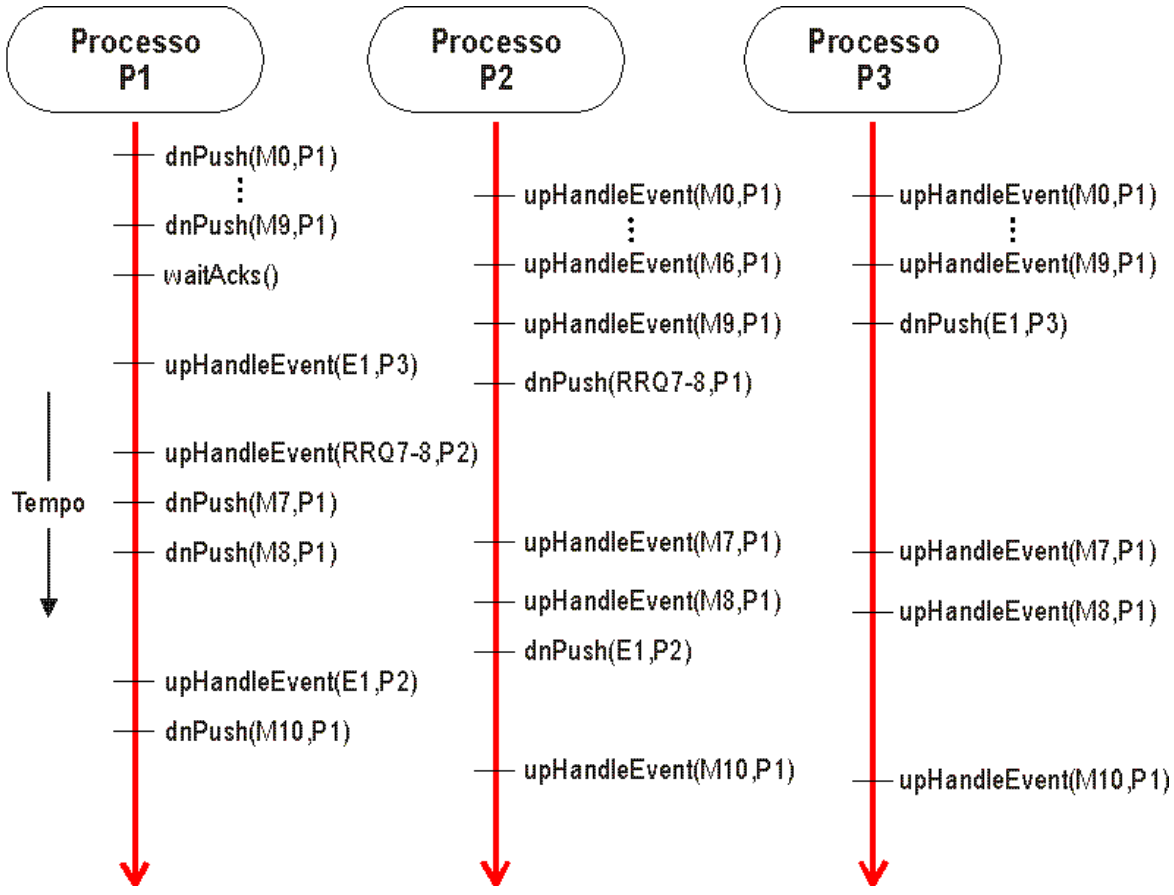

Figura 3.8 - Recuperação de mensagens pela camada NAK.

Na Figura 3.8 temos outro exemplo de recuperação de mensagens, porém neste exemplo não será necessário utilizar eventos de *HeartBeat*. O membro P1 envia as mensagens de M0 a M9. O membro P2 não recebe as

mensagens M7 e M8 devido a alguma falha e o membro P3 recebe todas as mensagens. A recuperação das mensagens perdidas por P2 será efetuada assim que ele receber uma mensagem com numSeq maior que o de M7 ou M8. Ao receber qualquer mensagem é feita uma comparação para verificar se o numSeq recebido corresponde ao esperado, ou seja, ao último numSeq recebido mais um. Se o numSeq não for equivalente ao esperado então a camada NAK presume que houve alguma falha no recebimento de mensagens e solicita o reenvio das mensagens supostamente perdidas.

Assim, quando a mensagem M9 é recebida por P2, então ele constata que perdeu algumas mensagens e envia uma mensagem solicitando a retransmissão de M7 e M8. O evento de retransmissão é recebido por P1 que atende ao pedido enviando as mensagens requeridas. Após receber M7 e M8, P2 constata que a época 1 está completa e envia a mensagem de Ack que permite que P1 inicie o envio da próxima época (M10).

É necessário que a camada NAK tenha conhecimento sobre quais os membros considerados ativos em um determinado instante, para que possa gerenciar o envio e recebimento de mensagens Ack. Estas informações são obtidas através do recebimento das visões geradas pela camada REACH que está configurada abaixo da camada NAK na pilha de camadas.

3.4.7 Camada REACH – Serviços Prestados

3.4.7.1 Geração de visões do grupo

Este serviço tem como objetivo manter um controle sobre quais membros compõem um grupo em um determinado instante. A cada alteração na composição do grupo são geradas visões de grupo que contêm a lista de todos os membros ativos naquele instante. Estas visões são utilizadas apenas localmente, pelas camadas superiores, não sendo enviadas ao grupo. A partir delas são providos serviços mais complexos como a recuperação de mensagens (NAK) e a visão atômica de grupo (TF-AM). A camada REACH tipicamente é utilizada acima da camada IPMCAST.

O mecanismo para geração de visões é baseado no envio de mensagens do tipo *HeartBeat* ao grupo. Portanto, todos os membros em intervalos regulares enviam mensagens *HeartBeat*. Elas têm o intuito de sinalizar aos outros membros

que o remetente continua ativo. Caso, um membro falhe, não enviará *HeartBeats,* logo os outros membros ao notar que não estão recebendo as mensagens *HeartBeat* esperadas deverão gerar visões excluindo o membro do grupo que falhou.

Existem dois parâmetros muito importantes que devem ser definidos quando da criação da camada REACH, eles são: **`heartBeatInterval`** e **`timeout.`** O primeiro representa o período para o envio dos *HeartBeats* ao grupo e o segundo qual o período máximo que um membro poderá ficar sem receber eventos *HeartBeat* de outro membro, antes de gerar uma nova visão o excluindo do grupo*.*

Portanto, quando um *HeartBeat* é recebido pelo membro local, ele verifica se o remetente da mensagem está inserido no grupo. Caso não esteja, é gerada uma nova visão incluindo o membro no grupo. A partir de sua inclusão é esperado que mensagens *HeartBeat* sejam recebidas periodicamente. Quando não for recebida nenhuma mensagem *HeartBeat* em um intervalo de tempo igual ou superior ao definido no atributo **`timeout`**, então deverá ser gerada e repassada para as camadas superiores uma nova visão do grupo excluindo o membro em questão. A verificação para identificar a falta de *HeartBeats*, com o intuito de excluir membros do grupo, é efetuada logo após ser enviado um evento *HeartBeat* ao grupo.

Na Figura 3.9 está apresentada uma situação onde são geradas novas visões de grupo devido à inclusão de um novo membro no grupo. No exemplo, o grupo P inicialmente é composto por dois processos distintos {P1, P2}. O **`heartBeatInterval`** definido é equivalente a 1500ms e o **`timeout`** a 2000ms. Podemos notar que P1 gera em t=0 um evento *HeartBeat* que é recebido por P2 que registra o recebimento, indicando que P1 está ativo. O membro P2 também gera eventos periódicos em t=800. No momento t=1200 o processo P3 é iniciado e logo em seguida (t=1300) envia o seu primeiro *HeartBeat*. Quando P1 e P2 recebem a mensagem de P3 eles reconhecem que um novo membro foi incluído no grupo e geram novas visões locais incluindo P3. Porém, P3 somente reconhece que P1 e P2 existem nos instantes t=1700 e t=2400, respectivamente quando eles geram mensagens de *HeartBeat*. Ao receber estas mensagens de *HeartBeat* P3 gera visões locais e no momento

t=2450, P3 reconhece todos os membros do grupo. É importante notar que P1 e P2 geram *HeartBeats* em intervalos regulares equivalentes ao **`heartBeatInterval`**, assim P1 gera eventos em t=0 e t=1500 e P2 gera em t=800 e t=2300.

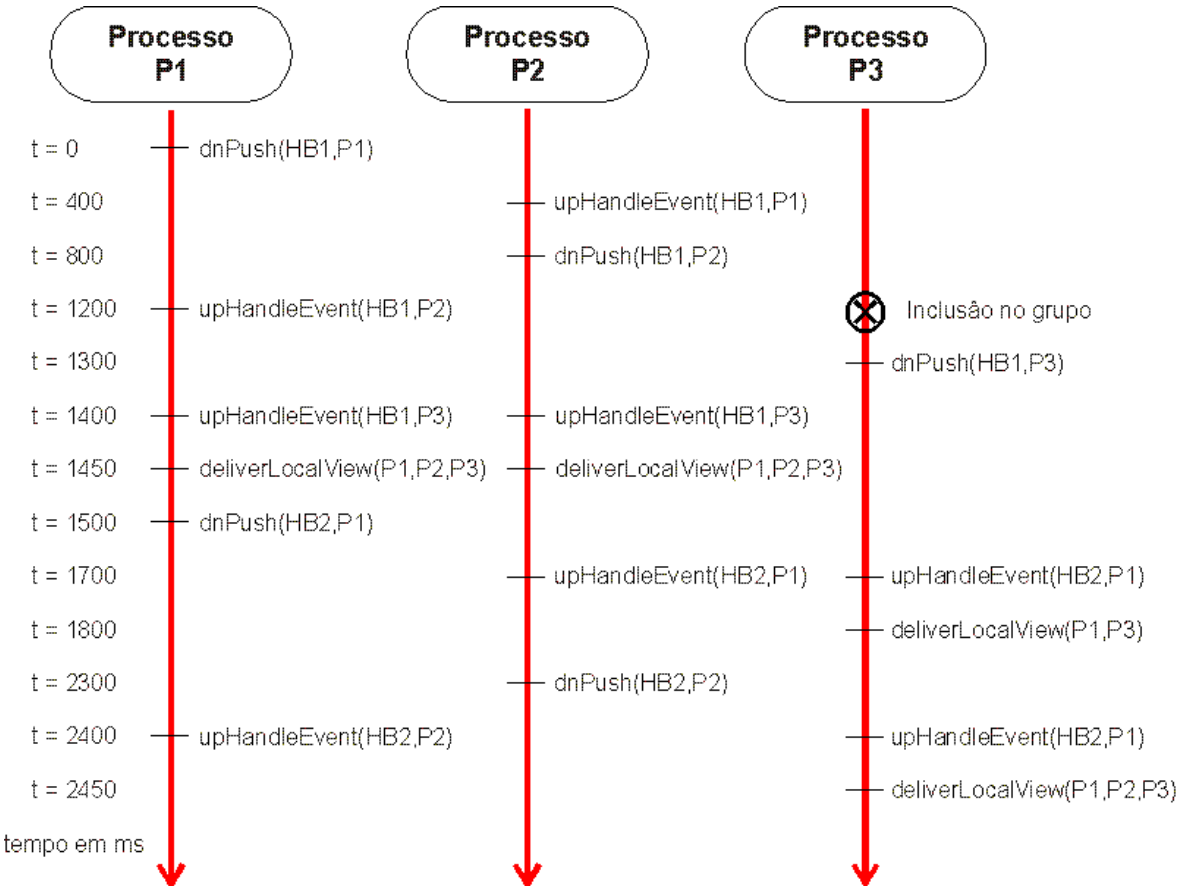

Figura 3.9 - Geração de nova visão de grupo devido à inclusão de um novo membro.

Outro exemplo onde são geradas visões de grupo é apresentado na Figura 3.10, porém desta vez o motivo é a exclusão de um membro do grupo (P3). Inicialmente o grupo P é composto pelos membros {P1,P2,P3}. Em t=0 P3 envia um *HeartBeat* que é recebido por P1 e P2 em t=200. Neste mesmo instante, o membro P3 apresenta uma falha e cessa a sua execução, devendo ser excluído do grupo. O membro P1 no momento t=800 envia um novo *HeartBeat* e em seguida verifica se há a necessidade de gerar um nova visão devido à exclusão de algum membro. Ele pesquisa qual foi a última mensagem recebida dos membros do grupo. No caso de P3, foi recebida uma mensagem em t=200, como o **`timeout`** definido é de 2000ms, então não é gerada uma nova visão. O membro P2 envia um *HeartBeat* em t=1000 e também verifica que não é

necessário gerar uma nova visão. A exclusão de P3 somente é percebida por P1 quando um novo *HeartBeat* é gerado em t=2300. Neste momento, P1 constata que a última mensagem recebida de P3 foi no instante t=200, com o **`timeout`** definido chega-se à conclusão que deveria ter sido recebida uma mensagem até o momento t=2200, como a verificação está sendo feita em t=2300 é gerada uma nova visão do grupo contendo apenas os membros P1 e P2. O mesmo processo ocorre com P2 no instante t=2500, quando é gerada uma nova visão local excluindo P3.

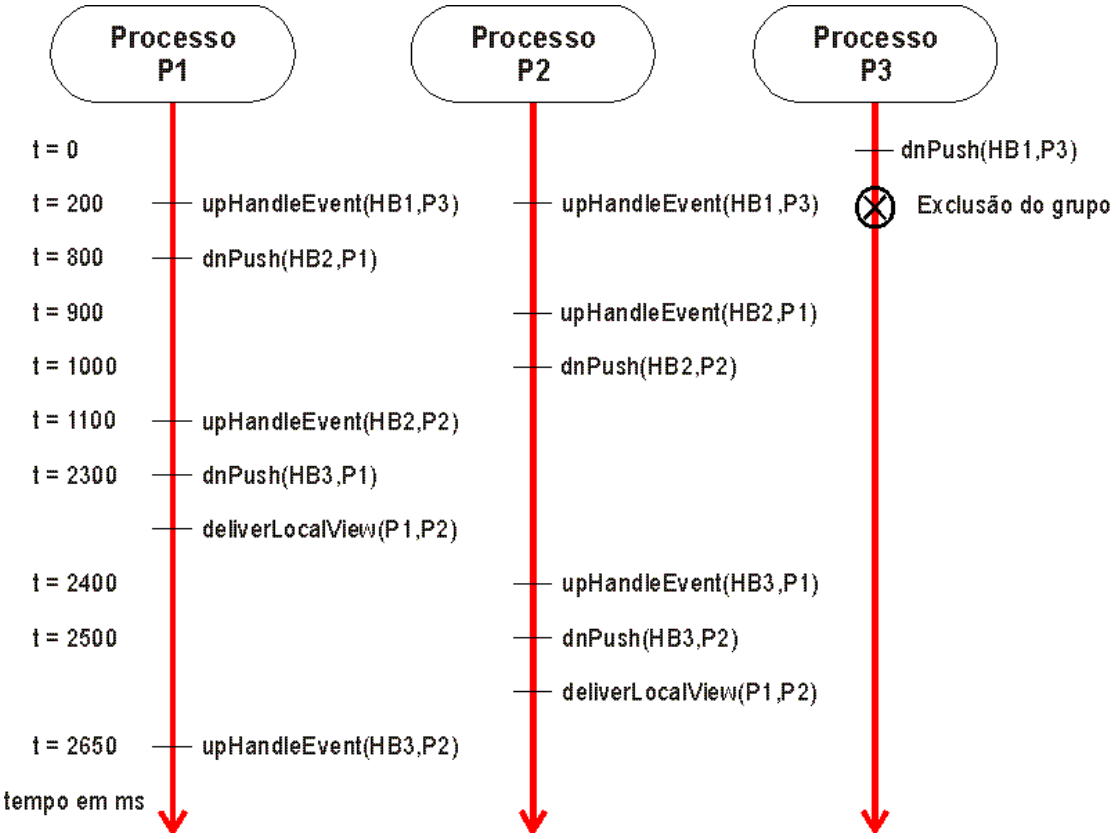

Figura 3.10 - Geração de visões de grupo devido à exclusão de um membro do grupo.

É importante notar que o mecanismo utilizado na camada REACH não provê qualquer espécie de controle de visão distribuída. Portanto, cada membro gera novas visões a qualquer tempo, fazendo com que exista a possibilidade de que diferentes membros de um grupo possuam visões do grupo divergentes, esta limitação é tratada pela camada TF-AM, que provê um serviço de controle de membros atômico que faz com que todos os membros convirjam para ter uma única visão instalada (vide seção 3.4.7.1).

Outra limitação da camada REACH é o fato de que podem ser geradas visões de grupo falsas, que excluem membros que continuam ativos. Isto é possível, pois as visões geradas são baseadas exclusivamente no recebimento de mensagens do tipo *HeartBeat.* Porém, elas podem ser perdidas ou mesmo entregues com atraso, provocando a falsa impressão que um membro falhou. Estes fatos podem ocorrer porque o serviço de comunicação utilizado é assíncrono e não confiável (protocolo UDP).

3.4.8 Camada IPMCAST – Serviços Prestados

3.4.8.1 Envio e recebimento de mensagens

Esta camada é responsável por enviar e receber as mensagens geradas pela aplicação e pelas camadas superiores para o grupo de processos ou para um único membro. O serviço de comunicação provido não é confiável, assume que as camadas superiores realizem a fragmentação de mensagens e não garante o tempo de resposta.

Esta é a camada mais baixa da pilha do iBusTF, por conseguinte, não possui uma camada imediatamente inferior. Abaixo da camada IPMCAST encontra-se a rede de comunicação, com a qual ela interage para enviar e receber as mensagens de membros remotos. A camada imediatamente acima na pilha de camadas é a REACH.

Para prestar o serviço de envio e recebimento de mensagens, é utilizado o protocolo UDP (*Unicast*) para mensagens ponto-a-ponto e IP *Multicast*, para mensagens com destino a um grupo de processos. No envio de mensagens a membros remotos deve ser tomada a decisão sobre qual dos dois serviços, *Unicast* ou *Multicast*, deve ser utilizado. A camada IPMCAST consegue discernir sobre qual o tipo de transmissão deverá ser efetuada através da avaliação do endereço destinatário, contido nas mensagens que recebe da camada NAK para envio. Quando o endereço é o de um grupo *Multicast*, ou seja, um endereço IP da classe D (varia entre 224.0.0.1 e 239.255.255.255), então deve ser utilizado o IP *Multicast*. Quando o endereço em questão for de um membro do grupo, deve ser feita a comunicação ponto-a-ponto.

Quando mensagens são recebidas de um grupo, é feita uma verificação para identificar se o membro local está inscrito no grupo cuja mensagem foi recebida, caso não esteja então as mensagens são descartadas.

3.5 Sumário

Neste capítulo apresentamos um breve resumo sobre os protocolos de comunicação especializados. Em seguida detalhamos uma FCG, baseada no modelo em camadas, chamada de iBusTF, descrevemos sucintamente seu histórico e enfatizamos os serviços prestados por cada camada que a compõe.

Os serviços do iBusTF que foram descritos na seção 3.4 serviram de base para a construção de outra FCG, baseada no modelo de canais de eventos, chamada de iBusTFE. A seguir, no capítulo 4, detalharemos o iBusTFE.

4 Uma Ferramenta de Comunicação em Grupo Baseada em Canais de Eventos

4.1 Introdução

O iBusTFE é uma FCG estruturada internamente de acordo com o modelo baseado em canais de eventos. Ele foi desenvolvido com o objetivo de prover as mesmas funcionalidades que o iBusTF (vide capítulo 3). A principal diferença entre estas duas FCGs é a forma como elas estão estruturadas internamente, o iBusTF foi construído com base no modelo em camadas e o iBusTFE no modelo baseado em canais de eventos (vide capítulo 2).

Apesar de proverem as mesmas funcionalidades, as duas FCGs são totalmente divergentes no que se refere ao projeto e implementação, devido a sua forma de estruturação interna. A única classe Java que foi utilizada na implementação das duas FCGs foi a *Newtop,* que implementa em ambas FCGs a ordenação total de mensagens. Neste capítulo iremos apresentar detalhes sobre a arquitetura geral do iBusTFE.

O iBusTFE foi projetado para que cada camada do iBusTF tivesse um componente correspondente no iBusTFE, com exceção das camadas FRAG e IPMCAST, como detalharemos adiante. Os serviços prestados por uma camada são providos por um componente e as entidades que ele agrupa. Desta forma, será possível observar como os serviços prestados pelas camadas do iBusTF foram implementados no iBusTFE.

Assim, consideramos que a descrição dos serviços providos pelo iBusTFE foi realizada no capítulo 3, nas seções de “serviços prestados”. Esta abordagem foi adotada, pois ambas FCGs prestam os mesmos serviços às aplicações, não sendo necessário descrevê-los novamente neste capítulo. Quando for necessário faremos as devidas referências.

Este capítulo está organizado da seguinte maneira. Na seção 4.2 será apresentada a arquitetura geral do iBusTFE, iremos descrever qual a relação

entre o iBusTFE e o framework[12] EVA. Em seguida, será realizada uma breve explanação a respeito do EVA e então apresentaremos uma visão macro do iBusTFE com todos os seus componentes, entidades e fluxos de eventos. Serão apresentados os detalhes de projeto que influenciaram todos os módulos do iBusTFE.

4.2 Arquitetura geral

O iBusTFE foi construído com base em um framework chamado de EVA (*EVent-based Architecture*) [BGH+a00]. Este framework foi escolhido porque segue o modelo baseado em canais de eventos (vide capítulo 2) e provê funcionalidades básicas que facilitaram a construção do iBusTFE. Desta forma, foi possível aumentar a produtividade na implementação e obter uma FCG estruturada internamente de acordo com os objetivos deste estudo.

O EVA é composto por diversas classes Java que representam, entre outros elementos, as três abstrações centrais do modelo baseado em canais de eventos: entidades, eventos e canais de eventos[13]. O framework disponibiliza alguns tipos diferentes de entidades e eventos. Eles provêem os serviços que são requeridos com maior freqüência pelos programadores para construir aplicações. Maiores detalhes sobre o EVA serão apresentados na seção 4.2.1.

Podemos afirmar que o EVA funcionou como a base para a construção do iBusTFE, porém foi necessário estender suas funcionalidades para que o iBusTFE pudesse prestar as mesmas funcionalidades que foram providas pelo iBusTF. Os serviços providos pelo EVA foram incorporados ao iBusTFE através do mecanismo de herança, ou seja, foram construídas novas classes Java que herdam os métodos e atributos das classes do EVA e implementam as funcionalidades complementares requeridas.

Na Figura 4.1 está representado um exemplo típico de como o iBusTFE incorporou as funcionalidades providas pelo framework EVA. Podemos observar, de acordo com a notação UML, que a classe *GroupView* herda todos os métodos da classe *EventImpl* (parte do framework EVA). Porém, foi necessário adicionar à

---

[12] O termo framework é utilizado nesta dissertação representando um conjunto de classes que provêem funcionalidades que facilitam a construção de uma classe específica de programas.

[13] No EVA os canais de eventos são chamados de componentes.

classe *GroupView* mais dois atributos e cinco métodos para que ela possa prestar o papel que lhe cabe dentro da implementação do iBusTFE.

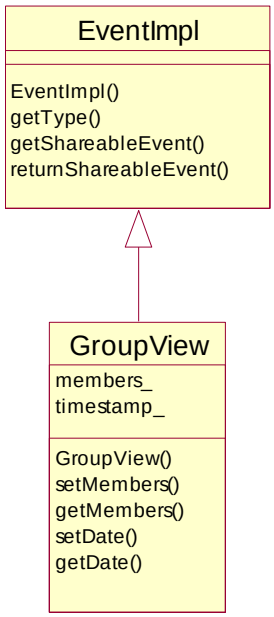

Figura 4.1 – Mecanismo de herança utilizado pelo iBusTFE para incorporar as funcionalidades básicas providas pelo EVA.

4.2.1 Framework EVA

O framework EVA [BGH+a00] é composto por um conjunto de classes Java que tem como objetivo prover funcionalidades tipicamente requeridas para desenvolver protocolos de comunicação especializados. Estas funcionalidades são incorporadas às aplicações através da utilização do mecanismo de herança [HOR97]. Assim é possível, conforme as necessidades do desenvolvedor, estender ou reimplementar determinadas funcionalidades.

Uma aplicação construída com EVA é composta por um conjunto de objetos que cooperam entre si trocando mensagens, chamadas de eventos. Estes objetos são os componentes e/ou entidades, cada componente pode agrupar uma ou mais entidades.

Os componentes além de estruturarem os sistemas, coordenam o envio de eventos entre as entidades de forma totalmente desacoplada. As entidades devem se registrar previamente junto a um ou mais componentes para consumir e/ou produzir eventos. Portanto, os componentes mantêm uma relação com todas

as entidades que registraram interesse em consumir eventos. Assim, é possível coordenar a entrega dos eventos produzidos.

O EVA foi concebido de acordo com o modelo baseado em canais de eventos. Sendo assim, ele possui classes que representam as três abstrações centrais deste modelo: entidades, evento e componentes. A seguir faremos um relato sobre elas:

Eventos – São objetos utilizados para carregar os dados que são enviados entre as entidades produtoras e as consumidoras de eventos. Um evento é um objeto que implementa a interface *Event*. O EVA provê a classe *EventImpl* que é uma implementação padrão de um evento. Portanto, tipicamente para se criar um novo tipo de evento basta criar uma classe descendente da classe *EventImpl* e agregar novos atributos e métodos de conforme necessário.

Existem outras duas interfaces importantes que se referem aos eventos: *ConsumableEventDescriptor e SupplyableEventDescriptor,* são utilizadas pelas entidades para definir como irão consumir e produzir os eventos, respectivamente. Por exemplo, quando uma entidade for se registrar junto ao componente para produzir um evento, ela deverá criar um objeto que implemente a interface *SupplyableEventDescriptor* e passar este objeto ao componente. O objeto criado tem a função de descrever os detalhes sobre a produção do evento (p.e.: qual o tipo de evento que será criado).

Entidades – As entidades são os segmentos do sistema que efetivamente fazem o trabalho, ou seja, todas as funcionalidades específicas da aplicação criada devem ser implementadas através da criação de entidades, para isso, elas interagem através da troca de eventos.

A classe *Entity* é a classe ancestral de todas as entidades, ela provê as operações básicas que todas as entidades executam. A partir dela foram implementadas algumas entidades que possuem características diferentes.

Há entidades ativas e passivas. As ativas permitem que os eventos que são recebidos possam ser consumidos de forma assíncrona, pois eles são armazenados em *buffers* para consumo futuro. Existe uma *Thread* independente para cada entidade ativa. Esta entidade é responsável por varrer os *buffers* da entidade que a criou, a procura de mensagens para serem consumidas.

Existe outro tipo de entidade que pode executar atividades de forma periódica, para isso, é necessário definir um intervalo de tempo e um método

quando da criação da entidade. Desta forma, o método criado será executado em intervalos regulares de tempo que correspondem ao intervalo estabelecido na criação da entidade. Este tipo de entidade é muito útil para, por exemplo, enviar eventos do tipo *HeartBeat* (vide seção 3.4.7).

Para implementar estas diferentes características, foram construídos oito tipos de entidades no EVA (vide tabela 4.1). Cada tipo provê mais de uma das características citadas concomitantemente. Por exemplo, a classe *TimedConsumerSupplierEntity* implementa uma entidade que pode produzir eventos, pode consumir eventos de forma assíncrona, pois é ativa, e pode executar uma atividade de forma periódica. Desta forma, os autores do EVA buscaram abranger os tipos de entidades tipicamente requeridos pelos desenvolvedores de aplicações.

Nome da classe	Características			
	Consumidora	Produtora	Ativa	Periódica
ConsumerEntity	X		X	
TimedConsumerEntity	X		X	X
TimedSupplierEntity		X	X	X
ConsumerSupplierEntity	X	X	X	
TimedConsumerSupplierEntity	X	X	X	X
CouplerEntity	X	X		
TimedCouplerEntity	X	X		X
ComponentEntity	X	X	X	X

Tabela 4.1 – Entidades implementadas no framework EVA e suas características.

As entidades que desejam consumir eventos devem registrar seu interesse junto a um ou mais componentes. Para executar o registro elas devem criar um objeto do tipo *ConsumptionDescriptor* e passá-lo para o componente, como parâmetro, através do método **`registerConsumptionEvent()`**. O *ConsumptionDescriptor* descreve como o evento deverá ser consumido. Entre outras informações, ele informa: qual o tipo de eventos a ser consumidos, passa uma referência ao objeto que irá consumir os eventos, indica qual o método deverá ser invocado para processar o evento (mecanismo de *callback)*, indica

qual a prioridade associada a este evento e se há um filtro para descartar eventos indesejados.

As entidades produtoras de eventos também necessitam criar um objeto para descrever como os eventos serão produzidos. Neste caso, são criados objetos do tipo *SupplyDescriptor* que são passados como parâmetros ao executar o método **`registerSupplyEvent()`**. Os objetos do tipo *SupplyDescriptor* são mais simples que os *ConsumptionDescriptor,* podendo conter apenas o tipo de eventos a ser produzido.

Componentes – Os componentes possuem duas funções principais: (1) estruturar a aplicação, agrupando entidades e (2) coordenar a troca de eventos entre elas.

É interessante notar que um componente também é uma entidade e pode se registrar junto a outro componente para consumo ou produção de eventos. Nestes casos, o efeito é como se todas as entidades subordinadas ao componente que se registrou junto a outro tivessem se registrado individualmente. Sendo assim, um componente pode ser tratado como uma caixa preta cuja interface é definida pelos eventos que consome e produz.

A classe *ComponentEntity* foi implementada no EVA para representar os componentes. Para que uma entidade possa enviar ou receber eventos para um componente ela deverá se subordinar a ele previamente, para isso, deve ser utilizado o método **`addEntity()`**.

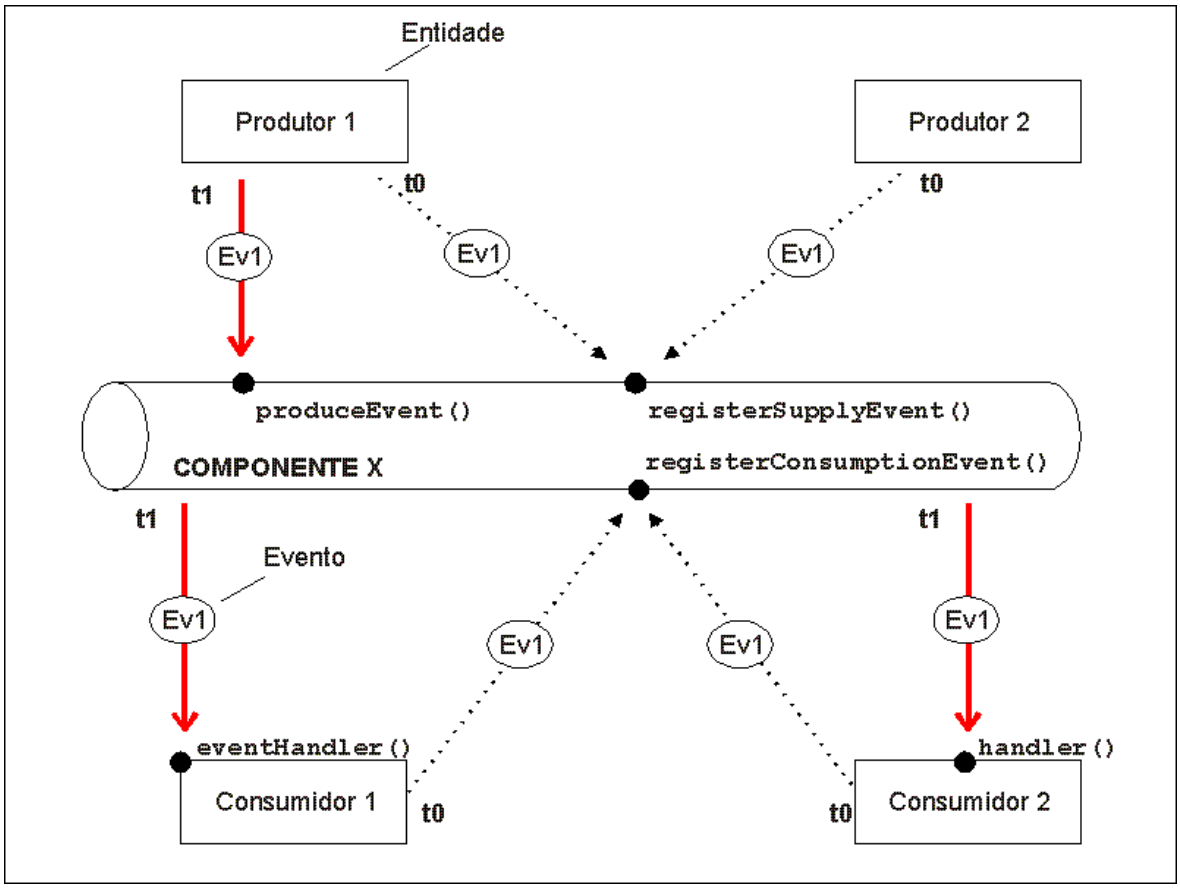

Figura 4.2 – Exemplo de aplicação típica construída com o EVA.

Na Figura 4.2 está apresentada uma aplicação típica construída com o EVA. No exemplo citado, existe um componente e quatro entidades que interagem entre si através do envio de eventos do tipo *EV1*. As entidades *Produtor1* e *Produtor2* se registraram junto ao componente *X* para produzir eventos no instante t0. Para isso, elas executaram o método **`registerSupplyEvent()`** e passaram como parâmetro um objeto que do tipo *SupplyDescriptor* que descreve qual o tipo do evento a ser produzido.

As entidades *Consumidor1* e *Consumidor2*, por sua vez, se registraram no instante t0 para consumir eventos do tipo *EV1*. Elas invocaram o método **`registerConsumptionEvent()`** e passaram um objeto do tipo *ConsumptionDescriptor* que tipicamente define o tipo do evento a ser consumido, sua prioridade, se existem filtros, passa uma referência ao objeto que irá consumir os eventos e indica qual o método que deve ser invocado.

No instante t1, a entidade *Produtor1* produz um evento do tipo *EV1,* Este evento é entregue às entidades que registraram para consumo. É interessante

notar que as duas entidades recebem o mesmo evento, porém o método utilizado para o recebimento possui o nome diferente. O *Consumidor1* recebe o evento através do método **`eventHandler()`** e o *Consumidor2* pelo método **`handler()`**. Isto é possível, pois o componente determina, através do *ConsumptionDescriptor* recebido no registro para consumo, qual método deve ser utilizado pelo mecanismo de *callback.*

Há outros três serviços prestados pelo EVA muito úteis para as aplicações em geral. São eles: (1) comunicação entre processos, (2) comunicação síncrona e (3) configuração dos componentes. Vamos descrever, sucintamente, cada um destes serviços a seguir:

Comunicação entre processos – O mecanismo apresentado até o momento permite que entidades se comuniquem através da troca de eventos, sob a coordenação de componentes. Porém, este mecanismo somente é possível se as entidades em questão estiverem no mesmo processo.

Porém, há uma gama crescente de aplicações que necessitam da comunicação entre processos, podemos citar o iBusTFE como um exemplo. Para resolver esta questão foram criados dois tipos especiais de entidades: *network notifier* e *network listener;* e um novo tipo eventos: *remote events.*

Para efetuar a comunicação entre processos, as entidades do tipo *network notifier* se registram para o consumo de eventos do tipo *remote event* junto a um ou mais componentes no processo remetente. Ao receber um *remote event*, estas entidades os enviam para os processos remotos através da rede de comunicação. No processo destinatário, as entidades do tipo *network listener* se registram para a produção de eventos juntos a um ou mais componentes. Os *network listeners* ficam escutando uma porta através da qual os eventos remotos são recebidos. Ao receber *remote events* de processos remotos, eles os entregam aos componentes em que se registraram para a produção de eventos. Desta forma, é possível que entidades localizadas em processos diversos se comuniquem. Os pares de *network listener* e *network notifier* são utilizados como uma ponte entre os processos que permitem que eventos remotos trafeguem.

O EVA provê três tipos diferentes de *listeners* e *notifiers: datagram unicast, datagram multicast* e *stream.* Os dois primeiros tipos provêem serviços de comunicação não confiável, sendo que o *datagram unicast* provê um serviço de comunicação entre duas entidades, ou seja, ponto-a-ponto e o *datagram multicast*

oferece um serviço de um-para-muitos. Finalmente, o *stream* oferece um serviço de comunicação ponto-a-ponto, confiável, com ordenação FIFO.

Comunicação síncrona – O EVA disponibiliza um mecanismo para que a comunicação síncrona possa ser realizada entre duas entidades distintas. É importante frisar que este mecanismo mantem o desacoplamento total que existe na comunicação assíncrona, pois ele também é coordenado pelo componente.

Para obter a comunicação síncrona os componentes permitem que entidades registrem "serviços" que elas disponibilizam de forma síncrona. Cada serviço registrado possui um identificador único junto ao componente. Assim, quando uma outra entidade for utilizar o serviço síncrono basta que ela execute o método **`invokeRequest()`** do componente em questão, passando como parâmetro o identificador do serviço e os parâmetros requeridos. O componente irá executar o método previamente registrado pela entidade que disponibiliza o serviço de forma síncrona. Vale ressaltar que a entidade que faz uso do serviço será bloqueada até que o método que implementa o serviço seja executado. É devolvido um objeto que contém a resposta ao método síncrono.

É necessário que a entidade que está disponibilizando um serviço se registre junto a um componente invocando o método **`registerService()`**. Através deste método a entidade deverá indicar o identificador do serviço, um objeto e o método que implementa que será executado quando alguma entidade necessitar do serviço.

Os componentes também podem registrar os serviços que prestam junto a outros componentes. Isto permite que as entidades que não utilizam um mesmo componente possam acessar serviços de entidades através de "supercomponentes".

Configuração de componentes – O EVA disponibiliza algumas classes que facilitam configurar a interação entre componentes, entidades e eventos, ou seja, configurar toda a aplicação construída. Através destas classes as aplicações que utilizam o EVA podem criar os componentes, registrar as entidades para consumo e produção de eventos e iniciar o funcionamento da aplicação. Estas classes implementam o padrão de projeto *Builder* [GHJV94].

É importante ressaltar que os protocolos de comunicação construídos com o EVA tipicamente servem como base para a construção de sistemas distribuídos. Como o desempenho destes sistemas está intimamente ligado ao

projeto do EVA, é extremamente importante que as questões que possam degradar o desempenho, como a sobre carga de tráfego, sejam amplamente estudadas.

Os autores de EVA tiveram a preocupação de analisar os “gargalos em potencial” existentes no framework e implementaram algumas técnicas para que eles fossem eliminados ou minimizados. Abaixo descrevemos estas técnicas:

- **Armazenando Consumidores** – Um componente pode se tornar um gargalo se existir um grande número de entidades ligadas a ele, pois desempenham um papel de intermediador, recebendo eventos dos produtores e os entregando aos consumidores. Este problema foi resolvido implementando-se um mecanismo onde as entidades produtoras enviam seus eventos diretamente para as entidades consumidoras.

  Para isso, é necessário que as entidades produtoras mantenham um controle sobre quais entidades registraram interesse em consumir os eventos que produz. Este fato poderia gerar um acoplamento indesejável, porém os componentes continuam assumindo o papel de coordenar a troca de eventos entre as entidades produtoras e consumidoras. Sendo assim, todo registro que é recebido para o consumo de eventos é notificado às entidades produtoras daquele evento, mantendo assim atualizada a relação de entidades consumidoras e eliminando a necessidade de passagem de referências a objetos de forma direta entre produtores e consumidores de eventos.

  Todas as entidades produtoras de eventos devem implementar a interface *Supplier* que define a operação **`consumptionNotification()`**. Através deste método os produtores de eventos são notificados sobre quais entidades desejam consumir eventos. Quando o componente recebe um novo registro ele invoca o método **`consumptionNotification()`**, passando uma referência para a entidade que deseja consumir eventos. A entidade produtora armazena estes registros para uso posterior. Desta forma, quando um novo evento for produzido ele será enviado diretamente aos consumidores, evitando que o componente em questão manipule os eventos.

  A Figura 4.3 apresenta um exemplo de como os eventos produzidos pela entidade *Produtor1* são enviados para duas as entidades

Consumidor1 e *Consumidor2*. Em t0 o *Produtor1* se registra para o envio de eventos *Ev1*. No instante seguinte, em t1 as entidades *Consumidor1* e *Consumidor2* se registram para o consumo de eventos *Ev1*. Ao receber o registro para consumo de eventos o componente *X* notifica o *Produtor1* sobre as inclusões na lista de entidades consumidoras. Assim, no momento t2 é produzido um evento *Ev1* e ele é encaminhado às entidades consumidoras diretamente.

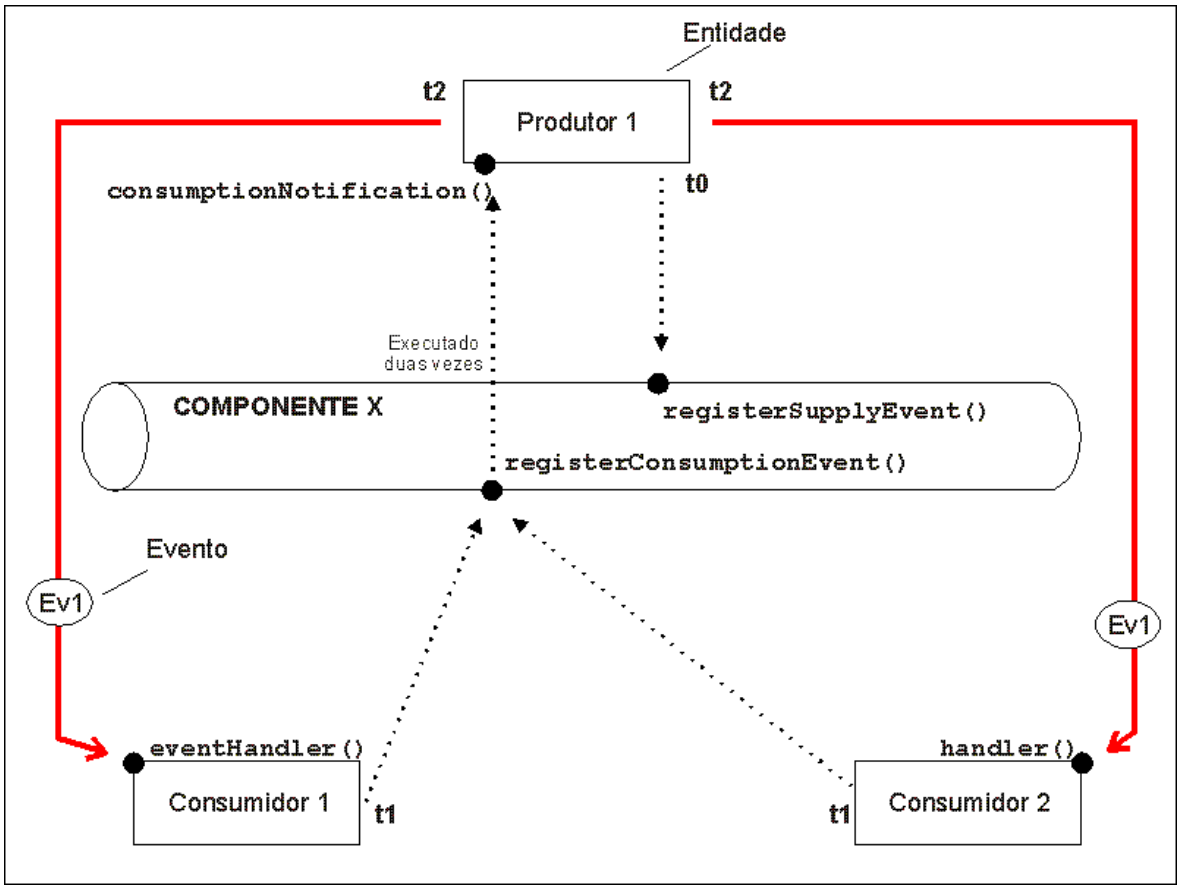

Figura 4.3 - Envio de eventos sem a intermediação do componente.

Note que a participação do componente no momento do envio das mensagens não existe. Assim, não é introduzido nenhum retardo no momento do envio da mensagem. O componente cumpre o seu papel quando do registro da intenção de consumo, a cada registro recebido ele invoca o método **`consumptionNotification()`** de todas as entidades que produzem o evento em questão, no caso apenas a entidade *Produtor1*. Desta forma, as entidades produtoras ficam com a relação atualizada de todos os objetos que devem consumir seus eventos e quais os métodos devem ser invocados para a entrega dos eventos.

- **Armazenando Serviços** – Os componentes também são responsáveis por gerenciar a prestação de serviços (síncronos). Eles desacoplam as duas entidades envolvidas em uma chamada, porém, este mecanismo gera chamadas desnecessárias ao componente. Para contornar este problema quando uma entidade necessita fazer uma chamada a um serviço ela pode utilizar o método **`getService()`** para recuperar uma referência à entidade que é implementa o serviço. Desta forma, caso o serviço em questão seja utilizado com freqüência, apenas será necessário executar o método **`getService(),`** uma única vez. O componente não precisa ser envolvido nas chamadas posteriores, pois a entidade que utiliza o serviço possui uma referência ao objeto que o implementa.
- **Geração de eventos otimizada** – Para produzir um evento é necessário criar uma instância da classe requerida, alocando uma área na memória. Este tipo de operação pode degradar o desempenho de aplicações que necessitam enviar e receber um número elevado de eventos. Aliado a este fato, um número elevado de aplicações cria os eventos para serem utilizados por um período curto de tempo o que potencializa a necessidade de alocação e liberação de áreas na memória. Para contornar este problema o EVA criou objetos que atuam como fábricas de eventos, chamadas de *Factory* (fábrica).

  Com a utilização das *Factories* quando uma entidade necessita de um evento ela solicita à fábrica correspondente. Todas as fábricas ao serem criadas instanciam um número pré-determinado de objetos dos eventos requeridos. Assim, elas possuem um estoque de eventos livres e quando necessário os disponibilizam para as entidades. Por sua vez, as entidades após utilizarem os eventos devem devolvê-los para sua fábrica, aumentando o número de eventos livres para consumo. Assim, não é necessário instanciar um objeto cada vez que um evento for produzido, pois as fábricas mantêm referências para todos os objetos alocados em memória que não estão sendo utilizados até o momento.

  As fábricas são criadas quando as entidades se registram para a produção de um evento[14]. Existe um objeto do tipo *FactoryPool* que

[14] Os eventos que são criados pelas *Factories* devem implementar a interface *Factorable.*

armazena todas as fábricas de uma determinada entidade. Quando a fábrica é criada, ela instancia um número pré-definido de objetos mantendo uma referência para eles. Se o número de objetos livres atingir um limite mínimo, então um novo lote é instanciado. Por outro lado, quando o número de objetos livres excede um limite máximo, então alguns objetos são liberados.

- **Compartilhamento de Eventos** – Um evento produzido poderá ser consumido por diversas entidades. Tipicamente, os eventos são utilizados para alterar o estado local da entidade ou para produzir novos eventos. Assim, os eventos entregues às entidades consumidoras são utilizados apenas para leitura, ou seja, seus dados não são alterados. Sendo assim, quando um produtor envia um evento ele executa o método **`getShareableCopy()`** que incrementa o contador de referências para este objeto. Portanto, quando um evento é entregue para uma entidade consumidora, na realidade apenas uma referência a um evento compartilhado é passada**.** As entidades devem executar o método **`returnShareableCopy()`** que reduz o valor do contador de referências, após terminarem o uso do evento. Assim, quando o contador associado a um evento indicar que nenhuma entidade está o utilizando, então o evento é devolvido a sua fábrica. Com a utilização deste mecanismo não é necessário executar diversas cópias dos eventos na memória, alocando recursos e consumindo tempo de forma desnecessária. As entidades podem executar cópias dos eventos quando for estritamente necessário.
- **Filtragem de eventos** – É comum entidades necessitarem consumir eventos de forma seletiva, ou seja, consumir eventos de um tipo que possuam as características desejadas. Por exemplo, a entidade *E*, subordinada a um componente *C*, se registra para o consumo de um evento do tipo *T*. Porém, ela apenas tem interesse em consumir eventos com o campo valor acima de 5000. Neste tipo de situação o componente *C* iria entregar todos os eventos do tipo *T* para a entidade *E*. Ela iria receber os eventos e avaliar se o valor para determinar se o evento deve ser processado ou simplesmente descartado. Esta situação insere um processamento desnecessário, pois todos os eventos seriam consumidos.

Para resolver este problema, foi criado um mecanismo que permite que sejam associados filtros ao registro de consumo, permitindo que eventos não desejados sejam descartados antes de serem enviados para a entidade em questão. Para isso, foi criada uma classe abstrata chamada de *Filter* que define o método *discardEvent()*. Para utilizar este mecanismo, as entidades devem fornecer um objeto que implemente a interface *Filter* no instante que se registram para consumir eventos. Assim, o componente antes de enviar o evento à entidade, executa o método *discardEvent()* passando o evento como parâmetro. Este método retorna um valor boleano que determinará se o evento deve ser entregue ou não à entidade consumidora.

- **Priorização do consumo de eventos –** Uma entidade ao registrar o seu interesse em consumir determinado tipo de evento pode indicar qual a prioridade consumo. Os eventos que possuem a mesma prioridade serão tratados com ordenação FIFO. Esta funcionalidade é útil em situações onde determinados eventos podem conter informações que agilizem, ou mesmo possibilitem descartar o tratamento de outros eventos.

4.2.2 Arquitetura

Nesta seção, iremos apresentar como o iBusTFE foi projetado, quais componentes e entidades que o formam e quais os tipos de eventos que foram criados. O objetivo desta seção é fornecer ao leitor uma visão global do iBusTFE. Maiores detalhes sobre o projeto e a implementação de cada componente poderão ser obtidos em [BBb03].

Como o iBusTFE é estruturado internamente de acordo com o modelo baseado em canais de eventos ele foi construído a partir de componentes e entidades que interagem entre si através da troca de eventos. Cada componente pode agrupar uma ou mais entidades. Em contrapartida, uma entidade pode estar ligada a mais de um componente. Logicamente as entidades possuem apenas um componente principal. Consideramos os outros componentes em que ela se registra como secundários. Este conceito possibilitou agruparmos as entidades logicamente, de acordo com as funcionalidades que elas provêem. Por exemplo, uma entidade que possui como componente principal o NAK, pode estar ligada a outros componentes, porém logicamente ela faz parte do grupo de entidades que

provêem as funcionalidades relativas ao componente NAK que são responsáveis por prover o serviço de recuperação de mensagens.

Além de servir para agrupar logicamente entidades, os componentes também servem para coordenar a comunicação entre elas. Podemos afirmar que os componentes “organizam” o ambiente para que as entidades efetivamente “façam” o trabalho. De fato, são as entidades que implementam as funcionalidades existentes no iBusTFE.

Para construir o iBusTFE utilizamos, sempre que possível, a segmentação definida no iBusTF. De modo geral, para cada camada do iBusTF existe um componente correspondente no iBusTFE. Podemos observar na Tabela 4.2 a relação de equivalência existente entre componentes do iBusTFE e as camadas do iBusTF. É importante lembrar que esta equivalência se restringe às funcionalidades providas, pois no que diz respeito ao projeto e à implementação, as camadas são radicalmente diferentes dos componentes.

Existem duas exceções à regra de equivalência entre camadas e componentes: (1) Não há um componente correspondente à camada FRAG, pois o EVA provê originalmente a única funcionalidade que esta camada implementa. Portanto não foi necessário criar um novo componente. (2) Existem dois componentes *compGroupSender* e *compGroupReceiver* que juntos são responsáveis por interagir diretamente com a rede de comunicação para enviar e receber mensagens, respectivamente. Estas funcionalidades são providas no iBusTF pela camada IPMCAST.

Camada do iBusTF	Componente do iBusTFE
STACK	AppiBusTFE
TF	CompTF
FRAG	-
FIFO	CompFIFO
NAK	CompNAK
REACH	CompREACH
IPMCAST	CompGroupSender CompGroupReceiver

Tabela 4.2 - Correlação entre componentes do iBusTFE e camadas do iBusTF

Cada componente do iBusTFE possui uma ou mais entidades ligadas a ele. Estas entidades se comunicam, através da troca de eventos, visando disponibilizar juntas os serviços requeridos. A comunicação entre as entidades é realizada exclusivamente através da troca de eventos. Para implementar esta comunicação é necessário que as entidades se registrem para a produção e/ou consumo de eventos junto aos componentes.

Foi necessário criar vários tipos de eventos. Estes eventos podem ser divididos em dois grupos: os eventos remotos e os locais. Os remotos herdam seu comportamento da classe *remoteEvent* e conseqüentemente, podem ser enviados entre diferentes processos que possuem espaço de endereçamento de memória diferente, tipicamente localizados em máquinas distintas. O outro grupo se refere aos eventos locais. Eles herdam suas características da classe *Event* e são utilizados exclusivamente dentro do processo onde foram criados. Portanto, eles não podem ser enviados pela rede de comunicação.

Na Tabela 4.3 estão relacionados todos os eventos que foram criados no iBusTFE, nela estão relacionados os identificadores dos eventos[15], utilizados para representá-los em figuras e tabelas, os nomes das classes criadas para representar os eventos, os nomes dos componentes principais dos eventos, e os nomes das classes ancestrais. Os eventos, a exemplo das entidades, estão ligados logicamente a componentes principais, apesar de poderem ser alterados e manipulados por outros componentes.

Id	Nome da Classe	Componente	Classe Ancestral
1	AppEvent	AppiBusTFE	RemoteEvent
2	AckEvent	compNAK	RemoteEvent
3	EpochConclusion	compNAK	RemoteEvent
4	LostMsgNotification	compNAK	RemoteEvent
5	RetransReply	compNAK	RemoteEvent
6	RetransReq	compNAK	RemoteEvent
7	SuccessfullAck	compNAK	EventImpl
8	GroupView	compREACH	RemoteEvent

[15] O identificador (Id) atribuído aos eventos trata-se de um número serial único utilizado para fins didáticos.

9	HeartBeatEvent	compREACH	RemoteEvent
10	FailConfirmation	compTF	RemoteEvent
11	RefuteSuspect	compNAK	RemoteEvent
12	Suspect	compTF	RemoteEvent
13	NullMessage	compTF	RemoteEvent
14	GroupStablishment	compREACH	EventImpl
15	MemberInfo	compREACH	RemoteEvent

Tabela 4.3 – Eventos utilizados no iBusTFE

Nas seções que se seguem apresentaremos diversas figuras onde estão representados componentes, entidades e eventos. Descrevemos abaixo alguns padrões que adotamos para facilitar a compreensão do leitor:

- Os componentes são representados por retângulos com o nome expresso na parte superior. Eles contêm as entidades que logicamente o consideram como componente principal.
- As setas indicam fluxos de eventos que são produzidos entre componentes e entidades ou entre componentes.
- Uma seta que possui seu inicio junto a borda de um componente indica que o componente está enviando ou recebendo eventos. Se a seta iniciar junto a borda de uma entidade, então ela está produzindo ou consumindo o evento diretamente. Maiores detalhes serão fornecidos ao longo desta seção.
- Todas as referências aos eventos expostos são realizadas com base no seguinte padrão: NSU+T. Onde NSU representa o identificador do evento (vide tabela 4.3) e T indica se o evento em questão foi produzido localmente ou se foi recebido de um evento remoto. Assim, é possível em uma mesma figura existirem dois fluxos de eventos indicados por 1R e 1L. Ambos fluxos representam o envio de eventos *AppEvent*, porém o primeiro tipo representa os eventos recebidos de processos remotos e o segundo representa os eventos produzidos pelo processo local.

Podemos distinguir duas formas distintas de envio de eventos (comunicação) entre as entidades: (1) quando as entidades produtoras e

consumidoras possuem o mesmo componente principal e (2) quando elas possuem componentes principais diversos.

Na primeira forma, o componente gerencia a troca de eventos entre as entidades que se registraram junto a ele. No segundo caso, quando ocorre o envio de eventos entre entidades que estão associadas a diferentes componentes, alguns procedimentos adicionais devem ser adotados. A seguir descrevemos os detalhes sobre os problemas e as soluções que foram utilizadas para implementar este tipo de comunicação entre as entidades na construção do iBusTFE:

Componente X Componente – Os componentes podem se registrar junto a outros componentes para consumir ou produzir eventos. Desta forma, todos os eventos de um determinado tipo que são produzidos em um componente podem ser enviados para os outros componentes que se registrarem para consumi-los, exatamente como se fossem entidades. A Figura 4.4 retrata um exemplo utilizado na construção do iBusTFE sobre este tipo de comunicação. A seta existente entre os dois componentes indica que há um fluxo de eventos entre eles, o número expresso acima da seta representa o tipo de evento a ser enviado. Neste exemplo, o componente *compGroupReceiver* se registrou junto ao componente *compReach* para o envio de eventos do tipo 9R. Sendo assim, todos os eventos do tipo 9R que forem produzidos no componente *compGroupReceiver* serão enviados para o componente *compReach*. Assim, é possível que a entidade *groupController* receba os eventos produzidos pela entidade *iBusTFEListener*, sem que ambas estejam subordinadas ao mesmo componente principal.

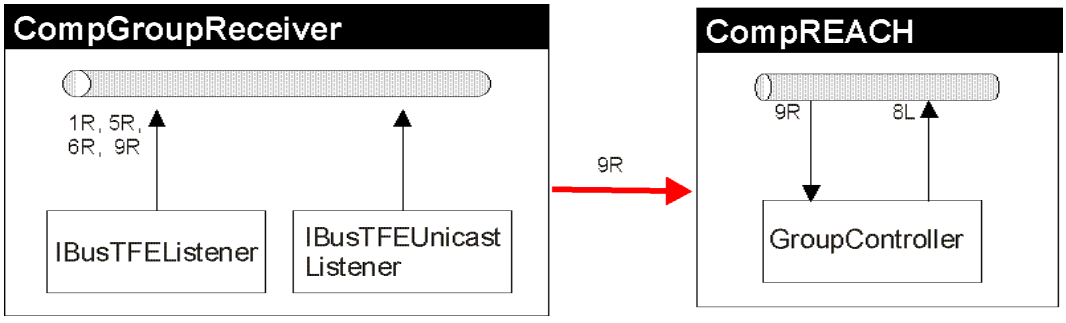

Figura 4.4 – Exemplo de envio de eventos “Componente X Componente”

A utilização desta forma de envio/recebimento de eventos é a mais recomendada, pois as entidades apenas necessitam se registrar junto ao seu componente principal, o que torna a configuração mais simples e elegante.

Porém, ela apresenta restrições que impossibilitam que ela seja utilizada de forma indiscriminada.

Na Figura 4.5 está representada uma situação onde o envio de eventos “Componente X Componente” gera problemas como a entrega de eventos em duplicidade e o envio prematuro de eventos. No exemplo exposto, a entidade *compGroupReceiver* envia eventos do tipo 1R e 5R para o componente *compFifo*. Por sua vez, o *compFifo* envia eventos do tipo 1R para o componente *compTF*. Até este ponto não há qualquer problema, porém, a entidade *Orderer* que está subordinada ao *compFifo*, necessita atualizar o evento 1R antes de enviá-lo ao *compTF*. Sendo assim, ela consome os eventos, os altera e produz o evento a ser enviado. Logo, há uma duplicidade na entrega dos eventos do tipo 1R.

A duplicidade ocorre porque todos os eventos recebidos são encaminhados para as entidades e componentes que registraram interesse em consumi-los. Assim, ao receber o evento 1R o *compFifo* o envia para o *compTF*. Porém, este evento será produzido novamente no *compFifo* após a entidade *Orderer* alterá-lo. Então, o evento 1R será entregue novamente ao *compTF*.

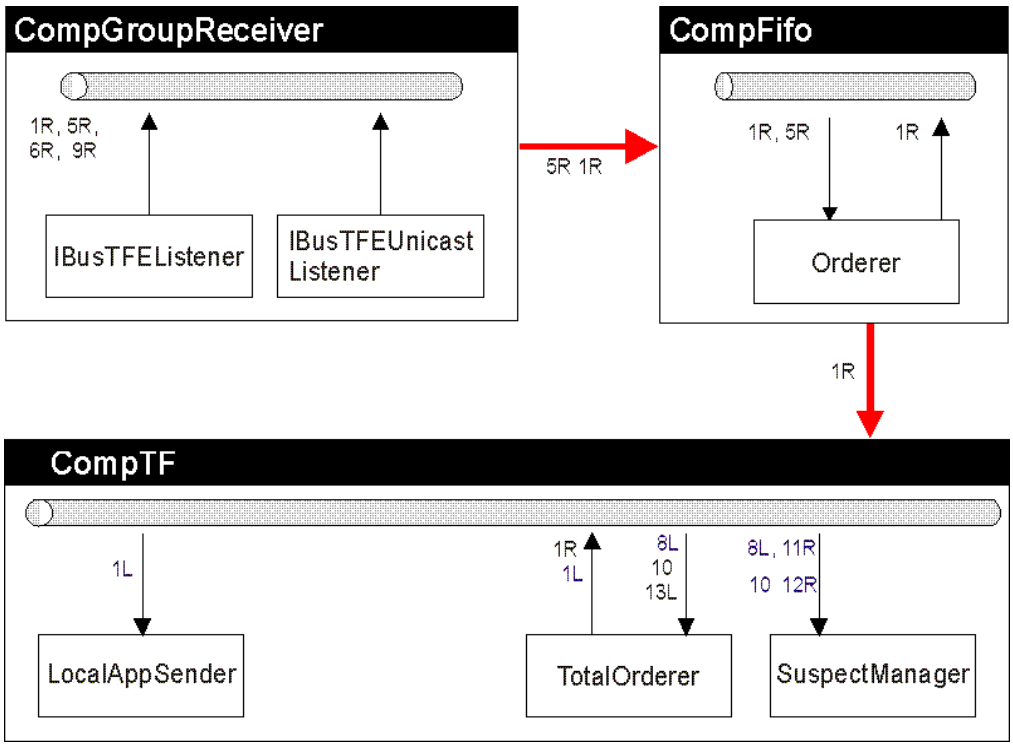

Figura 4.5 - Uso inadequado do envio de eventos entre componentes resultando em duplicidade no envio.

O problema reside na entrega do evento 1R pelo *compFifo* antes de sua alteração pela entidade *Orderer*. Na implementação utilizada do EVA não existe uma forma de fazer um envio seletivo de eventos, ou seja, alguma espécie de filtro que possibilitasse restringir o envio de eventos para o *compTF*. Portanto, utilizamos outras abordagens para contornar esta restrição.

Uma alternativa para resolver o problema exposto na Figura 4.5 seria registrar o componente *compTF* junto ao componente *compFifo* para o consumo de eventos do tipo 1R, indicando um filtro que eliminasse os eventos que não tivessem sido tratados pela entidade *Orderer*. Neste caso seria necessário criar um indicador que permitisse identificar quais os eventos devem ser consumidos.

Apesar de ser factível, esta alternativa obrigaria que os eventos fossem consumidos pelo componente que se inscreveu para o consumo do evento, no caso o *compTF*, antes de os enviar para as entidades consumidoras. Este fato inviabilizaria a entrega de eventos diretamente do produtor para os consumidores conforme descrito na seção 4.2.2, item “Armazenando Consumidores”. Todos os eventos teriam que ser entregues ao componente, através da chamada a um método definido na inscrição para consumo, e então enviados para as entidades consumidoras.

Esta abordagem poderia transformar os componentes em gargalos em situações onde o iBusTFE fosse submetido a um tráfego intenso de eventos. Portanto, com o intuito de não inserir retardos, optamos por não utilizar este tipo de configuração na construção do iBusTFE.

Componente X Consumidor - Neste tipo de comunicação, uma entidade se registra junto ao componente principal para a produção de um tipo de evento e junto a um componente secundário para o consumo. Assim, ela pode receber o evento, processá-lo devidamente e então enviá-lo para o componente principal, conforme Figura 4.6. Este fato resolve o problema de duplicação de eventos presente na comunicação “Componente X Componente”.

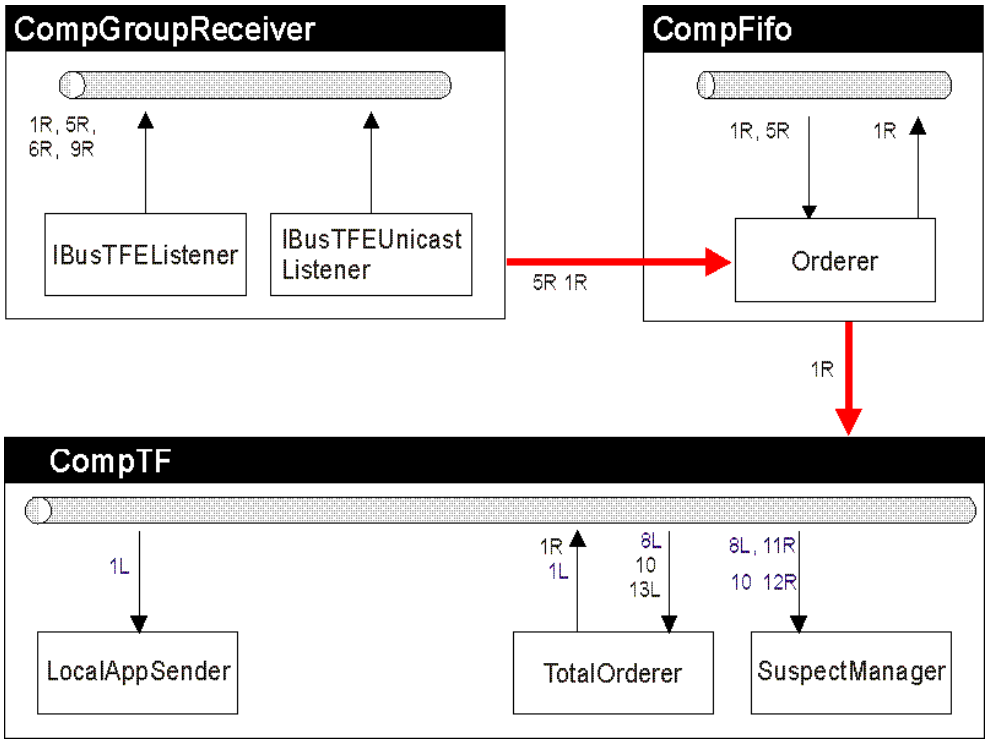

Figura 4.6 – Exemplo de envio de eventos “Componente X Consumidor”

Porém, esta abordagem também possui uma restrição grave: ela serializa o tratamento dos eventos. Portanto, seu uso é indicado nos casos onde naturalmente não há paralelismo, pois não existem duas entidades que tratam o mesmo tipo eventos de forma paralela.

Na Figura 4.7 está exposta uma situação onde a comunicação “Componente X Consumidor” foi utilizada de forma indevida. Podemos observar nela que as entidades *TotalOrderer* e *SuspectManager* tratam os eventos do tipo 8L. Este tratamento deve, preferencialmente, ser efetuado paralelamente com o intuito de maximizar o desempenho, a menos que alguma regra de negócio imponha o tratamento serial. No exemplo citado, não há restrições quanto ao paralelismo. Portanto, a utilização da comunicação “Componente X Consumidor” não é indicada para este caso, pois acarretará a serialização do tratamento dos eventos. Observe, que o componente *compReach* gera eventos do tipo 8L e os envia diretamente para a entidade *TotalOrderer.* Esta entidade trata o evento e o encaminha para o *compTF* que posteriormente o envia para a entidade *SuspectManager*. Portanto, o *SuspectManager* apenas iniciará o tratamento dos eventos após o *TotalOrderer* tratá-los, inserindo um retardo desnecessário. Neste

exemplo, a forma de comunicação mais indicada seria a comunicação “Componente X Componente”.

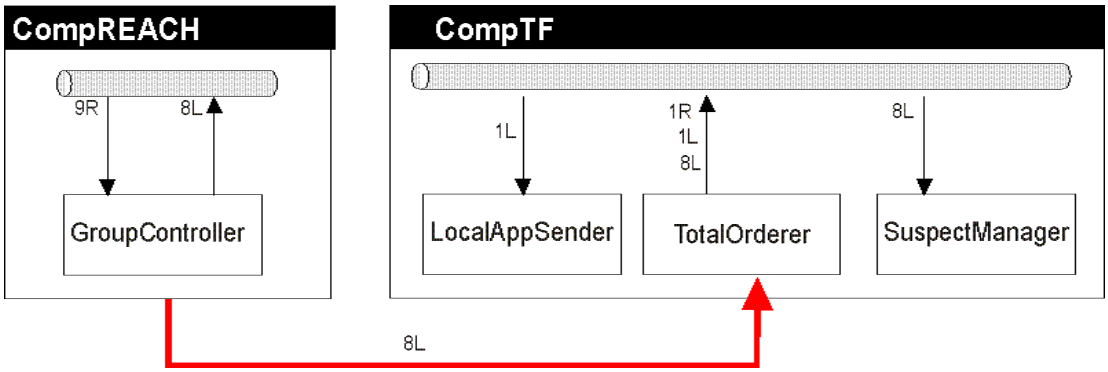

Figura 4.7 - Uso inadequado do envio de eventos entre componentes e um consumidor resultando no tratamento serializado dos eventos.

Produtor X Componente – Uma entidade pode se associar junto ao seu componente principal para consumo de um determinado tipo de evento e se associar junto a outro componente para produzir o mesmo tipo de eventos. Desta forma, a responsabilidade de produzir um evento em um componente remoto é transferida do componente para a entidade. Assim, quando um evento for enviado para o componente principal ele não o encaminhará para o componente destino, isto somente ocorrerá quando a entidade em questão produzi-lo. Esta abordagem permite que haja um controle mais rígido sobre quais eventos serão produzidos, eventos podem ser retidos, descartados ou alterados conforme for necessário. No exemplo exposto na Figura 4.8 a entidade *localAppSender* se registra junto ao componente *compTF* para consumo de eventos do tipo 1L e se registra junto ao componente *compNak* para produzir os eventos consumidos.

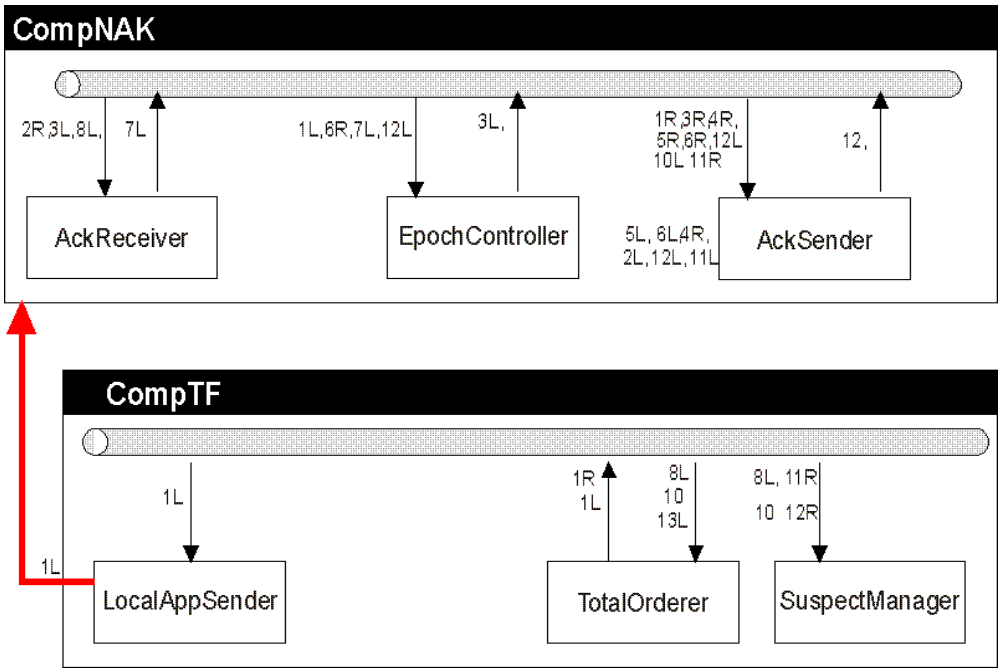

Figura 4.8 - Exemplo de envio de eventos "Produtor X Componente"

Este tipo de comunicação também impõem uma restrição, pois a implementação de EVA não permite que uma entidade possa produzir um mesmo tipo de eventos para mais de um componente.

A decisão de que tipo de comunicação deve ser utilizado é de extrema importância para que as aplicações construídas com o EVA funcionem corretamente e com o desempenho maximizado. Portanto, deve haver um entendimento claro sobre as virtudes e deficiências de cada tipo de comunicação entre entidades. Ao construir o iBusTFE, utilizamos os três tipos de comunicação apresentados, buscando maximizar as vantagens que cada tipo impõe e contornar suas limitações (vide Figura 4.9).

Algumas melhorias na implementação do framework EVA podem tornar mais simples a tarefa de projetar aplicações que o utiliza. Por exemplo, a implementação de um mecanismo para efetuar a entrega seletiva de eventos por parte dos produtores resolveria o problema da duplicação de eventos, permitindo que apenas a comunicação do tipo "Componente X Componente" fosse utilizada de forma indiscriminada encapsulando os detalhes para os desenvolvedores que utilizassem o EVA. Este fato tornaria o projeto mais simples e natural.

Na Figura 4.9, apresentamos o esquema geral do iBusTFE, nele estão representados todos os componentes, entidades e eventos criados. As setas

vermelhas (maiores) representam o fluxo lógico de eventos que são enviados entre componentes ou entre entidades e componentes secundários. As setas pretas (menores) indicam o envio de eventos entre as entidades e seus componentes principais. Os números que aparecem perto das setas correspondem aos eventos que compõem os fluxos lógicos. O mapeamento entre o nome dos eventos e seus números identificadores está descrito na Tabela 4.2.

Os detalhes sobre todos os componentes, entidades e fluxos de eventos que fazem parte do iBusTFE podem ser obtidos em [BBb03].

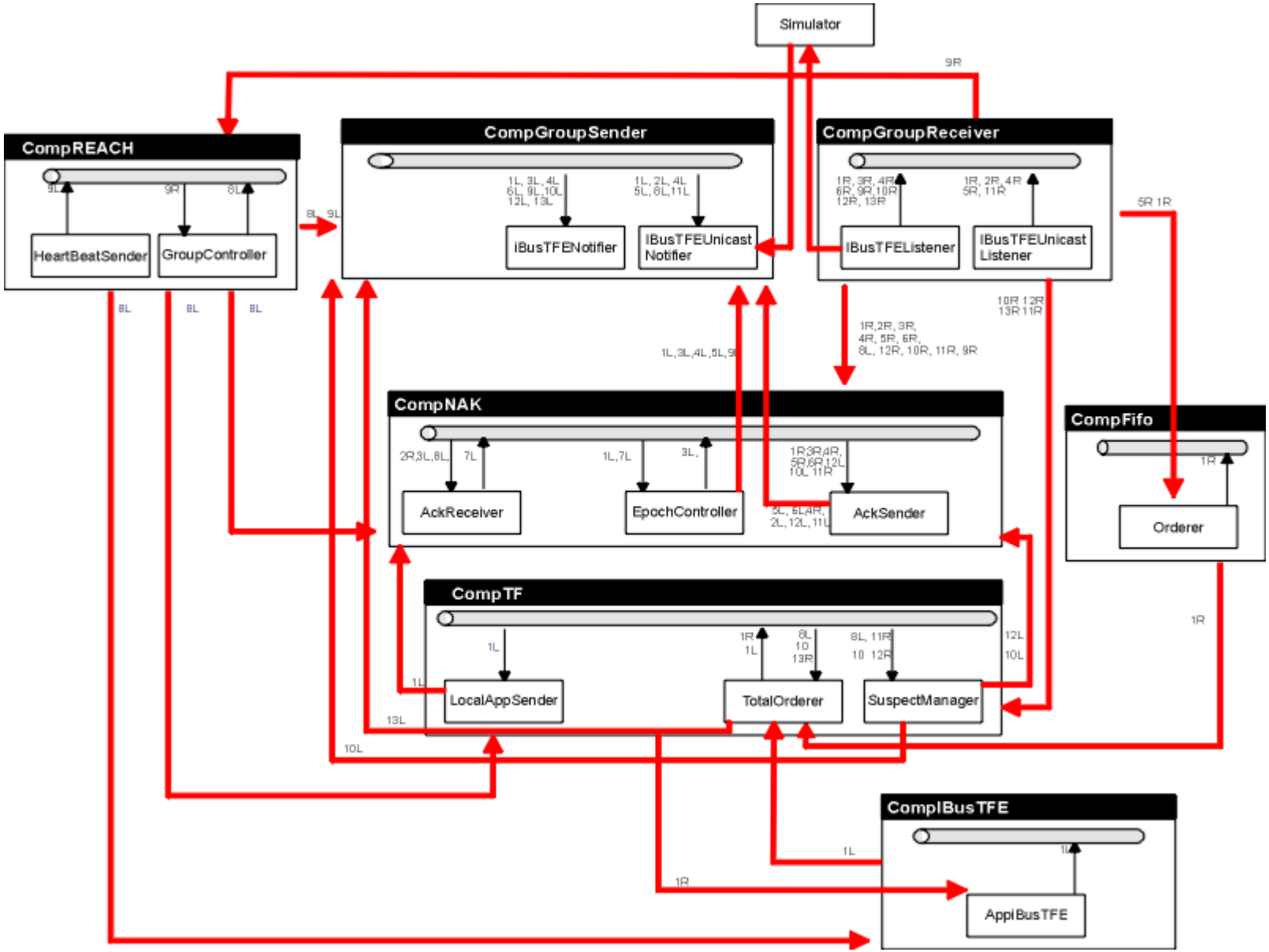

Figura 4.9 - Esquema geral do iBusTFE.

4.3 Sumário

Apresentamos neste capítulo uma visão geral do iBusTFE e determinamos como foram implementadas as funcionalidades, que originalmente foram providas pelo iBusTF. Sendo assim, é possível compararmos quais as diferentes abordagens adotadas para a construção do iBusTF e do iBusTFE. No próximo capítulo, apresentaremos os testes realizados com o intuito de determinar o desempenho das duas FCGs.

5 Avaliando o Desempenho das Ferramentas de Comunicação em Grupo

Neste capítulo iremos apresentar e analisar dados obtidos através da execução de testes com o iBusTFE e com o iBusTF a fim de demonstrar as vantagens e desvantagens do modelo em camadas e do baseado em canais de eventos, através de um estudo de caso, trazendo assim mais luz para a discussão sobre a utilização do modelo baseado em canais de eventos.

Ao construir o iBusTFE, tivemos a preocupação de isolar as variáveis que poderiam afetar a comparação, mantendo o foco da análise na diferença estrutural entre os modelos. A linguagem utilizada (Java) e as funcionalidades providas são as mesmas (vide capítulo 3). Desta forma, concentramos nossos esforços em avaliar qual foi o impacto causado pela forma de estruturação interna das FCGs avaliadas.

É importante ressaltar, que como as FCGs possuem formas de estruturação interna diversas, o seu projeto e implementação diferem radicalmente. Os detalhes de projeto e implementação do iBusTF e iBusTFE foram apresentados nos capítulos 3 e 4, respectivamente.

O critério utilizado para comparar as duas FCGs foi o desempenho, ou seja, a performance medida em unidades de tempo para enviar e receber dados entre processos remotos. Para obter os dados comparativos foram efetuados diversos testes utilizando uma pequena aplicação geradora de consensos [GHO^{+}00]. Esta aplicação foi construída exclusivamente para este fim e permitiu utilizarmos o iBusTF e o iBusTFE para enviar e receber mensagens de/para um grupo de processos remotos.

A análise sobre o desempenho das duas FCGs foi efetuada com o auxílio de uma ferramenta de perfilamento (Quantify – Rational [RAT02]). Assim, foi possível analisar não somente os tempos para obtenção dos consensos, mas também o tempo execução de cada método, dos seus métodos descendentes e a quantidade de chamadas, entre outros fatores estudados e discutidos neste trabalho.

Apresentaremos neste capítulo a metodologia que foi utilizada para obter os dados utilizados na avaliação das FCGs. Também descreveremos quais testes foram realizados, qual o ambiente utilizado e quais as ferramentas auxiliares foram utilizadas para mensurar e avaliar os resultados.

Serão discutidas as limitações encontradas nas duas FCGs, identificando falhas de projeto, erros de implementação e as restrições impostas pelo modelo em que se basearam, que são o foco principal deste trabalho. As limitações encontradas são analisadas de forma comparativa neste capítulo.

Organizaremos o restante deste capítulo da seguinte forma. Mostraremos na seção 5.1 qual foi a aplicação utilizada para testes com as duas FCGs. A seguir na seção 5.2, apresentaremos qual a metodologia utilizada nos testes. Entendemos que a metodologia abrange o ambiente computacional onde os testes foram executados e a forma como os dados foram obtidos, ou seja, número de testes, pontos de coleta dos dados etc. Na seção 5.3, apresentaremos os dados dos testes e faremos uma análise a respeito dos fatores que influenciaram o desempenho e apresentaremos as falhas de projeto identificadas nas FCGs.

5.1 Aplicação utilizada para testes

Foi construída uma aplicação chamada de App_Consensus que utiliza uma FCG para se comunicar com outros membros de um grupo. Esta foi projetada para utilizar tanto o iBusTF quanto o iBusTFE como FCG. A motivação para construí-la foi a de permitir mensurar o desempenho das duas FCGs, assim utilizamos uma aplicação comum em todos os testes realizados. Ao especificar os requisitos desta aplicação, tentamos mantê-la o mais simples possível, com o objetivo facilitar a análise dos resultados obtidos, concentrando as atenções nas duas FCGs utilizadas.

A *App_Consensus* é uma aplicação de consenso [GHO^{+}00], que permite a um grupo de processos remotos elegerem um valor consensual, para isto é realizada uma rodada de consenso, onde todos os membros do grupo indicam um valor sugerido para o consenso. A partir do conjunto de valores sugeridos é escolhido o menor valor entre eles. Note que a escolha do valor consensual somente pode ser realizada após o recebimento das mensagens sugeridas por todos os membros do grupo. Cada membro do grupo executa este algoritmo simples localmente e ao final do consenso todos devem eleger o mesmo valor.

O papel das FCGs é fundamental para que a *App_Consensus* possa implementar o mecanismo de consenso apresentado. Os processos utilizam dois serviços distintos das FCGs: (1) o envio/recebimento de mensagens um-para-muitos e (2) o recebimento de visões de grupo (vide seção 3.4.1). O primeiro serviço é

utilizado para enviar e receber as mensagens com os valores sugeridos para o consenso e o segundo serviço para poder determinar quais membros do grupo estão ativos.

O controle da visão do grupo é necessário, pois o algoritmo determinístico, que elege o valor de consenso, apenas pode ser executado após o recebimento das mensagens contendo os valores sugeridos de todos os membros ativos do grupo.

Na Figura 5.1, podemos observar o esquema geral da *App_Consensus* para a escolha de um valor consensual. Notem que no instante t0 as aplicações de consenso localizadas nos computadores A, B e C enviam para o grupo, utilizando uma FCG, os valores sugeridos para o consenso, no exemplo citado os valores sugeridos para o consenso são: 3, 2 e 1, respectivamente.

As FCGs dos computadores A e C, recebem todas as mensagens com os valores sugeridos (1, 2 e 3) até o momento t1. Assim, elas repassam as mensagens para a *App_Consensus.* Ao receber as mensagens, a aplicação de consenso determina que todas as mensagens enviadas pelos membros remotos foram recebidas e então escolhe o menor valor entre elas (instante t2). Este é o valor consensual eleito para a rodada de consenso, no exemplo citado na Figura 5.1, o valor consensual foi equivalente a 1.

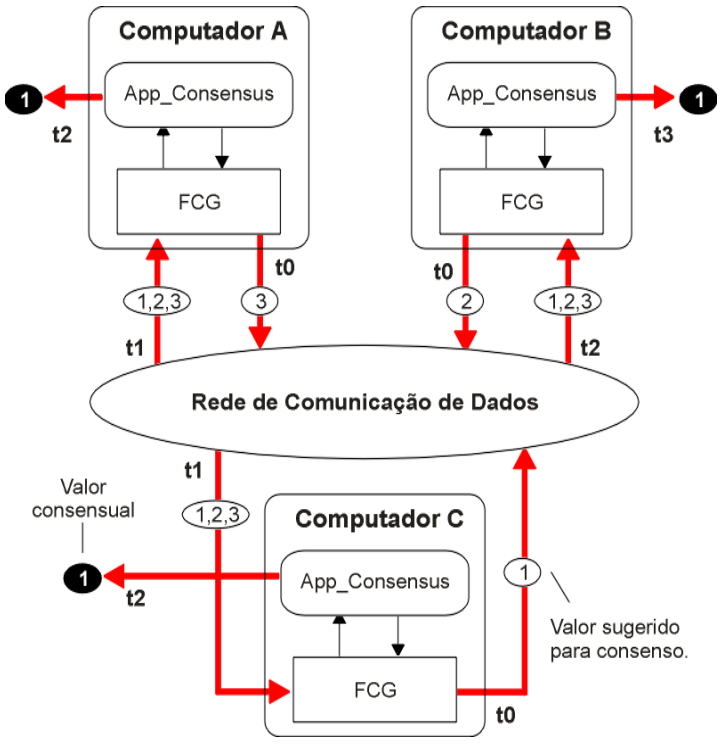

Figura 5.1 – Funcionamento da aplicação de consenso.

O processo para a escolha do valor consensual também irá ocorrer com a *App_Consensus* localizada no computador B. Porém, o valor consensual em B somente é obtido no momento t3. Isto pode ocorrer em função da rede de comunicação assíncrona que foi utilizada. Se as mensagens com os valores sugeridos forem entregues aos membros do grupo em instantes diferentes, haverá uma desincronia na obtenção do valor consensual pelos membros do grupo.

Cabe ressaltar que a aplicação de consenso não garante em que momento será obtido o valor consensual. A garantia provida é a de que todos os processos que fazem parte do grupo irão, em algum momento, eleger o mesmo valor consensual.

Para implementar a *App_Consensus*, foi construída a classe *Consensus*, onde foi disponibilizado o método **`produceSyncConsensus(),`** com o intuito de permitir que sejam realizadas múltiplas rodadas de consensos. Este método possui dois parâmetros: **`initialValue`** e **`loops`**. O primeiro representa o valor que será sugerido na primeira rodada de consenso e o segundo parâmetro, o número de rodadas de consenso que devem ser realizadas. Quando mais de uma rodada de consenso é requerida, então o valor sugerido em rodadas de consensos subseqüentes é o valor utilizado na rodada anterior adicionado a mais uma unidade.

O método **`produceSyncConsensus(),`** permite que sejam realizadas várias rodadas de consenso com apenas uma chamada. Porém, ele apenas envia a mensagem da rodada posterior quando a rodada atual for encerrada (valor consensual obtido). Assim, consideramos que o método **`produceSyncConsensus()`** envia as mensagens com valores sugeridos de forma síncrona, ou seja, o envio do valor sugerido para a próxima rodada de consenso fica bloqueado até que a rodada vigente seja finalizada.

É necessário informar quantos processos irão compor o grupo inicialmente. Esta informação é utilizada para bloquear o início do envio das mensagens. Portanto, se uma aplicação criar uma instância da FCG e executar o método para o envio das mensagens, enquanto o grupo inicial não tenha sido formado, as mensagens serão bloqueadas até que seja estabelecida a visão inicial do grupo.

5.2 Metodologia para os Testes

Para determinar o desempenho do iBusTF e do iBusTFE, executamos três grupos de testes. Cada grupo foi composto de diversas baterias de testes com a aplicação de consenso descrita na seção 5.1. Para isso, foi necessário preparar um ambiente adequado para a execução dos referidos testes, a fim de garantir que os dados obtidos fossem relevantes para este estudo.

Nossa preocupação foi a de não permitir que fatores alheios aos testes pudessem interferir nos resultados, dificultando a sua análise ou mesmo os tornando inválidos. Nossos principais focos de atenção com relação ao ambiente dos testes foram os seguintes:

- Garantir que no momento dos testes apenas os processos necessários estivessem ativos nos computadores utilizados.
- Diminuir a concorrência pelos recursos do sistema operacional no momento dos testes para facilitar a análise dos resultados.
- Isolar os computadores envolvidos nos testes de tráfego de rede alheio.
- Garantir que os computadores envolvidos nos testes possuíssem a mesma capacidade de processamento, ou seja, a mesma configuração no que se refere ao equipamento e aos programas utilizados (*hardware* e *software*).

Descreveremos na seção 5.2.1 detalhes de como preparamos o ambiente de testes e como pudemos garantir que ele estava pronto para a execução dos testes. Na seção 5.2.2, mostraremos como os testes foram realizados e qual a metodologia utilizada para determinar o desempenho da aplicação de consensos utilizando o iBusTFE e o iBusTF.

5.2.1 Ambiente de testes

Nosso primeiro esforço foi no sentido de reduzir o número de processos ativos durantes os testes. Para isso, analisamos a lista de serviços[16] ativos e identificamos quais não eram relevantes para os testes com a aplicação de consensos. Deixamos ativos apenas aqueles que julgamos serem necessários para o correto funcionamento do sistema, de modo geral, mantivemos ativos apenas

[16] Um serviço no sistema operacional Windows 2000 é implementado por um ou mais processos. Assim, reduzindo o número de serviços ativos conseguimos reduzir o número de processos.

alguns serviços do sistema operacional. Na Figura 5.2 apresentamos todos os processos que implementavam os serviços que permaneceram ativos durante os testes. Ressaltamos que o conjunto de serviços e processos ativos foi exatamente o mesmo em todas as máquinas envolvidas nos testes.

Como em todas as máquinas envolvidas nos testes o sistema operacional instalado foi o Windows 2000 Professional, pudemos utilizar o gerenciador de tarefas [MCa01] e a ferramenta de controle de serviços [MCb01] disponibilizados por este sistema operacional para identificar e parar os serviços e processos não relevantes para os nossos testes.

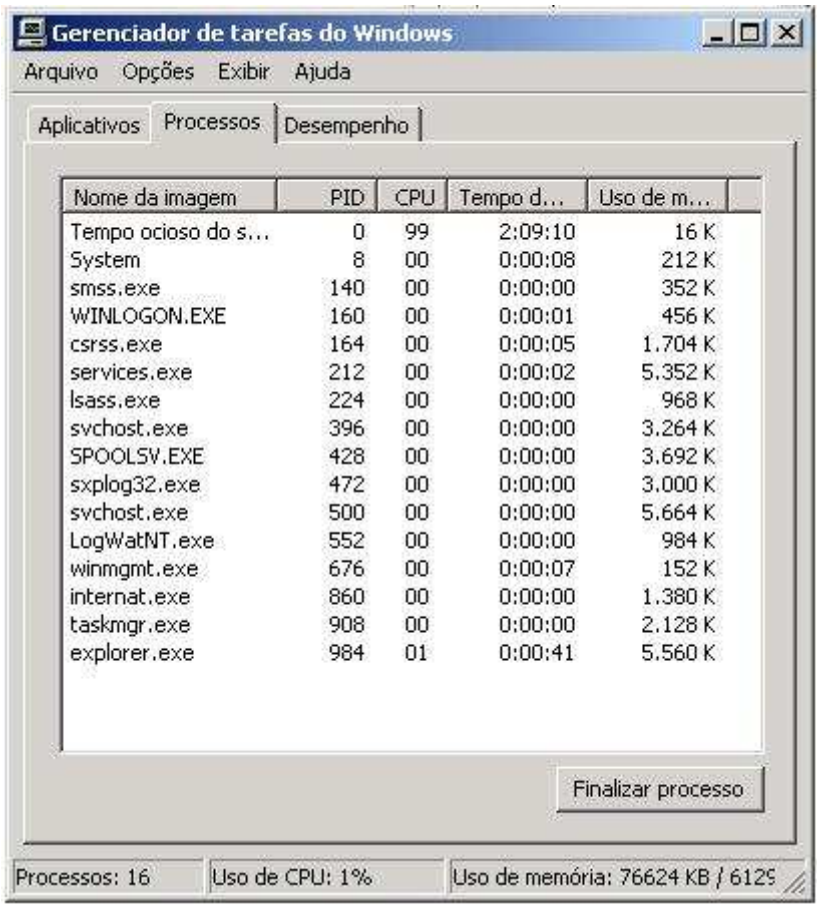

Figura 5.2 – Processos ativos nos computadores utilizados nos testes.

Outro fator estudado e controlado foi o tráfego de rede alheio aos testes, principalmente no que se refere ao envio de pacotes de broadcast na rede de comunicação utilizada. Este tipo de tráfego poderia interferir na realização dos testes interrompendo o sistema operacional. Para isolar os computadores criamos uma rede composta apenas pelas máquinas envolvidas nos testes. Estes computadores foram interligados através do uso de um switch da marca 3com modelo Super Stack

II 1100 (ref.3c16951) de 10-100 Mbits por segundo [TAN96]. Desta forma, apenas o tráfego gerado pelos computadores locais circulava pela rede de comunicação.

Todas as máquinas foram configuradas para utilizar o TCP/IP como protocolo de comunicação padrão e foi atribuído a cada uma delas um endereço IP fixo.

Também nos preocupamos em ter um grupo de máquinas semelhantes no que se refere ao hardware e ao software. O nosso intuito foi o de evitar que uma máquina, por ter um desempenho inferior se tornasse um gargalo nos testes, afetando os tempos obtidos negativamente.

Com este intuito, utilizamos um grupo formado por 6 máquinas todas com a mesma especificação de hardware e software (vide Figura 5.3) . No momento dos testes elas possuíam exatamente os mesmos grupo de processos ativos.

Configuração de Hardware:	
Processador:	Pentium III (Intel) 800MHz
Memória RAM:	250Mbytes
Placa de rede:	10Mbits/segundo
Configuração de Software:	
Sistema Operacional:	Windows 2000 Professional
Máquina virtual java:	Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0-C)

Figura 5.3 – Configuração das máquinas utilizadas nos testes com a aplicação de consenso.

A uniformização das máquinas, o controle dos serviços e processos e a criação de uma rede independente, permitiram que o ambiente de testes atendesse aos requisitos necessários para a execução dos testes previstos com a aplicação de consensos e as FCGs. Para constatar este fato, utilizamos uma ferramenta disponibilizada pelo Windows 2000 chamada de “Microsoft System Monitor” [MC00] para a análise do ambiente preparado para os testes. Procuramos verificar se após as alterações efetuadas, o ambiente estava apropriado para a execução dos testes.

Com o Microsoft System Monitor criamos um log de desempenho que a cada segundo colhia amostras sobre a utilização de vários recursos do sistema operacional, para cada recurso monitorado foi criado um contador. A relação completa dos contadores utilizados e dos recursos monitorados por eles pode ser vista na Tabela 5.1.

Contador	Descrição sobre o recurso monitorado
Pacotes de monodifusão (*Unicast*) recebidos por segundo	Pacotes de monodifusão (*Unicast*) recebidos por segundo é a freqüência com que se estão distribuindo pacotes de monodifusão (*Unicast*) a um protocolo de nível superior.
Pacotes de monodifusão (*Unicast*) enviados por segundo	Pacotes de monodifusão enviados por segundo é a freqüência com que os protocolos de nível superior solicitam a transmissão de pacotes para endereços de monodifusão (*Unicast*). Esta freqüência inclui os pacotes que foram descartados ou não foram enviados.
Pacotes de não monodifusão enviados por segundo	Pacotes de não monodifusão enviados por segundo é a freqüência com que os protocolos de nível superior solicitam a transmissão de pacotes para endereços que não são de monodifusão (Não *Unicast*) (por exemplo, pacotes de difusão [broadcast] ou difusão limitada [*multicast*]). Esta freqüência inclui os pacotes que foram descartados e os que não foram enviados.
Pacotes de não monodifusão recebidos por segundo	Pacotes de não monodifusão recebidos por segundo é a freqüência com que se distribui pacotes que não são de monodifusão (não *Unicast*) (por exemplo, pacotes de difusão [*broadcast*] ou difusão limitada [*multicast*]) a um protocolo de nível superior.
Total de bytes por segundo	Velocidade com que bytes são enviados e recebidos na interface de rede, incluindo os caracteres de

	quadros.
Memória\Páginas por segundo	Páginas por segundo é o número de páginas lidas de ou gravadas em disco para resolver falhas de página de hardware. (As falhas de página de hardware ocorrem quando um processo requer código ou dados que não estão em seu conjunto de trabalho ou estão em outro local na memória física e devem ser recuperados de disco). Este contador foi projetado como um indicador primário dos tipos de falhas que causam demoras em todo o sistema. Ele inclui páginas recuperadas para satisfazer a falhas no cache do sistema de arquivos (geralmente solicitadas por aplicativos) e arquivos de memória mapeada não armazenados em cache. Este contador mostra a diferença entre os valores observados nas duas últimas amostragens, dividida pela duração do intervalo de amostragem.
Processador\Porcentagem tempo de interrupção	Porcentagem de tempo de interrupção é a porcentagem de tempo transcorrido que o processador utiliza para tratar Interrupções de hardware durante o intervalo de amostragem. Este valor é um indicador indireto da atividade de dispositivos que geram interrupções, como o clock do sistema, o mouse, drivers de disco, linhas de comunicação de dados, placas de interface de rede e outros dispositivos periféricos. Esses dispositivos normalmente interrompem o processador ao concluírem uma tarefa ou quando necessitam de atenção. A execução normal de segmentos é suspensa durante as interrupções. A maioria dos clocks do sistema interrompe o processador a cada 10 milissegundos, criando um segundo plano de atividade de interrupção. Este contador mostra o

	tempo médio ocupado como uma porcentagem do tempo de amostragem.
Processador\Interrupções por segundo	Interrupções por segundo é o número médio de interrupções de hardware que o processador recebe e processa por segundo. Este valor é um indicador indireto da atividade de dispositivos que geram interrupções, como o clock do sistema, o mouse, unidades de disco, linhas de comunicação de dados, placas de interface de rede e outros dispositivos periféricos. Esses dispositivos normalmente interrompem o processador quando concluem uma tarefa ou necessitam de atenção. A execução normal de segmentos é suspensa durante as interrupções. A maioria dos clocks de sistema interrompe o processador a cada 10 milissegundos, criando um segundo plano de atividade de interrupção. Este contador mostra a diferença entre os valores observados nas duas últimas amostragens, dividida pela duração do intervalo de amostragem.
Processador\Porcentagem de tempo do processador	Porcentagem de tempo de processador é a porcentagem de tempo em que um processador está ocupado executando segmentos que não sejam ociosos. Este contador foi projetado como um indicador primário da atividade do processador. Ela é calculada medindo-se o tempo que o processador gasta na execução do segmento do processo ocioso em cada intervalo de amostragem e subtraindo-se esse valor de 100%. (Cada processador tem um segmento ocioso que consome ciclos quando nenhum outro segmento está pronto para ser executado.) Pode ser encarada como a porcentagem do intervalo de amostragem empregada em trabalhos úteis. Este contador mostra a porcentagem média de

	tempo ocupado observada durante o intervalo de amostragem. Ela é calculada monitorando-se o tempo em que o serviço esteve inativo e subtraindo-se esse valor de 100%.
Servidor\Total de bytes por segundo	Número de bytes que o servidor enviou e recebeu da rede. Este valor fornece uma estimativa global do grau de ocupação do servidor.
Duração da amostragem (minutos:segundos)	Intervalo de tempo em que os recursos do sistema operacional foram monitorados.

Tabela 5.1 – Contadores utilizados para analisar o consumo de recursos nos computadores envolvidos nos testes com a aplicação de consenso.

Após preparar o ambiente para os testes e antes de executá-los efetivamente, ativamos nos computadores a serem utilizados o log de desempenho. Verificamos que os resultados obtidos foram muito semelhantes em todas as máquinas envolvidas nos testes, isto se deve ao fato de que todas elas possuíam exatamente a mesma configuração (hardware e software) e de não haver no ambiente de testes nenhum processo externo à rede criada, influenciando as máquinas envolvidas. Na Tabela 5.2, estão expressos os valores médios obtidos ao longo do período estudado para todos os contadores mensurados.

CONTADORES	**MEMBROS DO GRUPO**					
	0	**1**	**2**	**3**	**4**	**5**
Pacotes de monodifusão (*Unicast*) recebidos por segundo	0,000	0,000	0,000	0,000	0,000	0,000
Pacotes de monodifusão (*Unicast*) enviados por segundo	0,000	0,000	0,000	0,000	0,000	0,000
Pacotes de não monodifusão recebidos por segundo	0,831	0,706	0,715	0,734	0,895	0,871
Total de bytes por segundo	47,828	40,857	40,856	41,658	52,318	51,314
Memória\Páginas por segundo	0,005	0,005	0,005	0,005	0,006	0,007
Processador\Porcentagem tempo de interrupção	0,019	0,009	0,009	0,015	0,055	0,026

Processador\Interrupções por segundo	105,793	105,416	105,642	105,303	106,247	105,711
Processador\Porcentagem de tempo do processador	0,122	0,042	0,047	0,048	0,647	0,140
Servidor\Total de bytes por segundo	0,000	0,000	0,000	0,000	0,000	0,000
Duração da amostragem (minutos:segundos)	29m12s	20m35s	15m38s	24m02s	19m44s	18m25s

Tabela 5.2 – Consumo de recursos do sistema operacional registrado nas máquinas utilizadas nos testes com a aplicação de consenso.

Os contadores de recebimento e envio de pacotes monodifusão (*Unicast*), indicaram que não havia nenhum processo local enviando ou recebendo mensagens alheias aos testes. Os contadores de envio e recebimento de pacotes de não monodifusão (*multicast* e *broadcast*) apresentaram taxas muito baixas que não interferem, significativamente, nos resultados dos nossos testes. Os valores registrados pelo contador de páginas por segundo também são baixos e indicam que os processos que permaneceram ativos não estavam executando tarefas que necessitassem carregar novos dados na memória.

Os contadores relativos ao percentual de tempo de interrupção e interrupções por segundo indicam que a maioria das interrupções estava sendo realizada por recursos do próprio sistema operacional, como o clock de tempo, o mouse e outros periféricos que geram interrupções. Este indicador foi muito importante, pois a partir dele pudemos constatar que as interrupções geradas por fatores alheios (p.e.: pacotes *broadcast* e *multicast*) aos testes foram minimizadas e que nos níveis apresentados não teriam influencia significativa nos testes. Finalmente, o último contador indicado demonstrou que nenhuma outra máquina estava enviando ou recebendo dados para os computadores.

Também medimos o percentual de tempo de disco, que indica qual a porcentagem de tempo que a unidade de disco rígido esteve ocupada atendendo a requisições de leitura ou gravação, utilizando um contador de forma transiente, ou seja, sem ser armazenado em disco (o que afetaria os resultados) os valores tenderam a 0%. Isto indicou que os processos que permaneceram ativos durante os testes não faziam acesso a disco. A memória disponível nos membros também foi

medida em todos os computadores e todos apresentaram valores similares como pode ser constatado na Tabela 5.3.

Nome Computador	Código do Membro	Memória disponível (Mbytes)
TREINO20	0	174
TREINO04	1	176
TREINO18	2	173
TREINO03	3	174
TREINO09	4	175
INSTRUTOR01	5	174

Tabela 5.3 - Memória disponível nos computadores utilizados na execução dos testes com a aplicação de consenso.

A memória física disponível nos computadores (vide Tabela 5.3), superou as necessidades da aplicação de consenso durante os testes, evitando o movimento de informações entre a memória e o disco rígido (swapping) [TAN95].

Sendo assim, constatamos que o ambiente de testes estava adequado para a execução e mensuração dos testes com a aplicação de consensos utilizando as duas FCGs estudadas nesta dissertação.

5.2.2 Descrição dos testes realizados e da metodologia para mensurar o desempenho

Para mensurar o desempenho das duas FCGs estudadas, executamos diversas baterias de testes utilizando a *App_Consensus*. Fizemos três grupos de testes, onde cada grupo foi composto por várias baterias de testes.

Inicialmente, previámos que seria necessário executar apenas um grupo de testes, porém ao executar o primeiro grupo os resultados obtidos não foram conclusivos, demandando a execução de testes adicionais. De modo geral, podemos afirmar que os grupos de testes subseqüentes foram realizados para eliminar fatores que estavam prejudicando a análise dos dados. Detalharemos na seção 5.3.1 os testes realizados, os resultados obtidos e quais foram as motivações para realizá-los.

Com este intuito, foram realizados testes com grupos de 3, 4, 5 e 6 membros. Em todos os testes, cada membro do grupo estava localizado em um computador distinto e foi identificado por um código que variou de 0 a 5. Foram realizadas 10 baterias de testes com cada FCG e com cada tamanho de grupo descrito, uma bateria de testes foi composta por 99 ou 500 rodadas de consensos. Em todas as baterias de testes, os membros foram iniciados sempre na mesma ordem (crescente com base no código do membro) e registramos os tempos para a obtenção de consensos de todos os membros envolvidos nas rodadas. Estes tempos foram gravados em um arquivo para cada bateria de testes.

Os dados utilizados para a análise do desempenho foram obtidos com a realização de consensos síncronos (vide seção 5.1). Esta abordagem foi adotada visando simplificar a análise final, pois assim minimizamos a degradação do desempenho, no momento dos testes, ocorrido devido à concorrência por recursos do sistema operacional.

No consenso assíncrono, tipicamente há a ocorrência do envio e do recebimento de mensagens de forma paralela. Isto ocorre, pois as duas FCGs possuem várias *Threads* [SUN02], que podem ler e enviar mensagens em paralelo. Quando as *Threads* estão sendo executadas elas disputam os recursos do sistema operacional. Normalmente, este problema é resolvido bloqueando, momentaneamente, uma das *Threads* que estão concorrendo pelo recurso do sistema operacional (p.e.: pelo processador). Este fato leva a uma degradação do desempenho e dificulta a análise proposta por este estudo, porém este problema foi minimizado, pois utilizamos apenas os consensos síncronos. Neste tipo de consenso, as mensagens com valores sugeridos para a próxima rodada de consenso apenas são enviadas após o recebimento de todas as mensagens da rodada de consenso anterior. Desta forma, conseguimos serializar o envio e recebimento de mensagens.

Lembramos que a utilização de consensos síncronos minimiza o problema da concorrência pelos recursos do sistema operacional, mas não o resolve definitivamente, pois no momento dos testes obrigatoriamente havia outros processos rodando nos computadores, conforme descrito na seção 5.2.1.

Também não consideramos, para efeito da análise final, os resultados obtidos com os testes onde mais de um membro se localizava em um mesmo

computador. Este fato é justificado, pois estes membros concorrem pelos recursos da máquina onde estavam instalados degradando o seu desempenho.

Para registrar o desempenho de cada FCG utilizamos inicialmente a seguinte métrica: medimos o período de tempo compreendido entre o envio da mensagem com o valor sugerido pela aplicação de consenso e a obtenção do valor consensual, após receber todas as mensagens dos membros do grupo relativas àquela rodada. Vale ressaltar que utilizando esta métrica, as marcas de tempo de início e fim do período encontrado foram obtidas através do relógio local do micro, eliminando erros devido à falta de sincronização entre os diferentes relógios dos computadores.

Com a evolução dos testes, sentimos a necessidade de adicionalmente utilizar outras métricas, assim, começamos a calcular o tempo para a obtenção de consensos de mais duas formas. Em ambas, consideramos que o consenso inicia quando o primeiro membro envia um valor sugerido para o consenso. O final do consenso, na primeira forma, é determinado quando o primeiro membro do grupo obtem o valor consensual e na segunda forma quando o último membro determina qual o valor consensual.

É importante ressaltar que para utilizar as duas formas de cálculo, citadas no parágrafo anterior, foi necessário determinarmos, antes da realização de cada bateria de testes, qual a desincronia inicial existente entre os relógios dos computadores envolvidos nos testes. O cálculo da desincronia foi fundamental, pois o tempo para início de um consenso poderia ser determinado pelo relógio de um computador, enquanto que o final do consenso poderia ser determinado por outro computador. Assim, para podermos comparar os valores obtidos a partir de relógios de máquinas distintas era obrigatório considerar a desincronia.

Nos dois grupos de testes iniciais calculamos o tempo para obtenção dos consensos apenas utilizando a primeira métrica citada (diferença entre os tempos de envio e de obtenção do consenso obtidos no mesmo relógio). No último grupo de testes utilizamos as três formas de calcular o tempo para obtenção dos consensos.

Para efetuar o cálculo da desincronia inicial existente entre os relógios dos computadores foram criadas duas classes que chamamos de *DesincCalc* e *TimeEchoServer*. Estas classes implementam um algoritmo simples que foi utilizado antes da execução de cada bateria de testes para efetuar o cálculo da desincronia.

Para isso, elegemos uma das máquinas envolvidas nos testes para fornecer a hora de referência e em todas as outras mantivemos o programa *TimeEchoServer* ativo. Então, a partir da máquina com tempo de referência (MR), executamos o *DesincCalc*, que lia o relógio da MR, obtendo o tempo t1. Em seguida era enviada uma mensagem, via socket, utilizando o protocolo UDP para outra máquina do grupo. Na outra máquina o programa *TimeEchoServer* recebia a mensagem de MR, lia o relógio local, obtendo o tempo t2, e devolvia uma resposta para a MR contendo o t2. A máquina de referência ao receber a mensagem contendo o tempo t2, lia novamente seu relógio e obtinha o tempo t3.

Com os tempos t1, t2 e t3 efetuamos o cálculo da desincronia entre a máquina de referência e a outra envolvida nos testes. A desincronia existente corresponde a t2 – t1 se t3 – t1 < 1ms for verdadeiro. Notem que t3 - t1 representa o tempo que a mensagem gastou para ser enviada pela rede, processada pelo membro remoto, e enviada a resposta pela rede (contendo t2). Para obter um valor preciso, este processo foi repetido até que o valor de t3 - t1 fosse menor que 1ms.

Calculamos a desincronia inicial entre os membros na inicialização de todas as baterias de testes, para cada membro participante. Os valores de desincronia obtidos foram armazenados nos arquivos de log e utilizados na análise dos dados.

Para verificar se a variação da desincronia durante os testes era representativa, construímos uma pequena aplicação chamada de *DesincCalcTest* que calculava 100 vezes a desincronia existente entre o computador de referência e os outros computadores do grupo. Entre duas medições a aplicação dormia durante 1000ms.

No Gráfico 5.1, está expressa a variação ocorrida durante uma medição utilizando o *DesincCalcTest.* Os tempos de desincronia inicial medidos foram –24ms, -56ms, 95ms e 14ms para os computadores TREINO04 (membro 1), TREINO18 (membro 2), TREINO03 (membro 3) e TREINO09 (membro 4), respectivamente. No Gráfico 5.1, observamos a variação ocorrida sobre os tempos inicialmente medidos. De modo geral, houve apenas uma pequena variação de 1 a 2ms no decorrer dos testes. Porém, registramos que esporadicamente havia algumas variações maiores (até 10ms), como pode ser observado no consenso 42 do membro 2, onde houve uma variação de –8ms em relação à desincronia inicial medida.

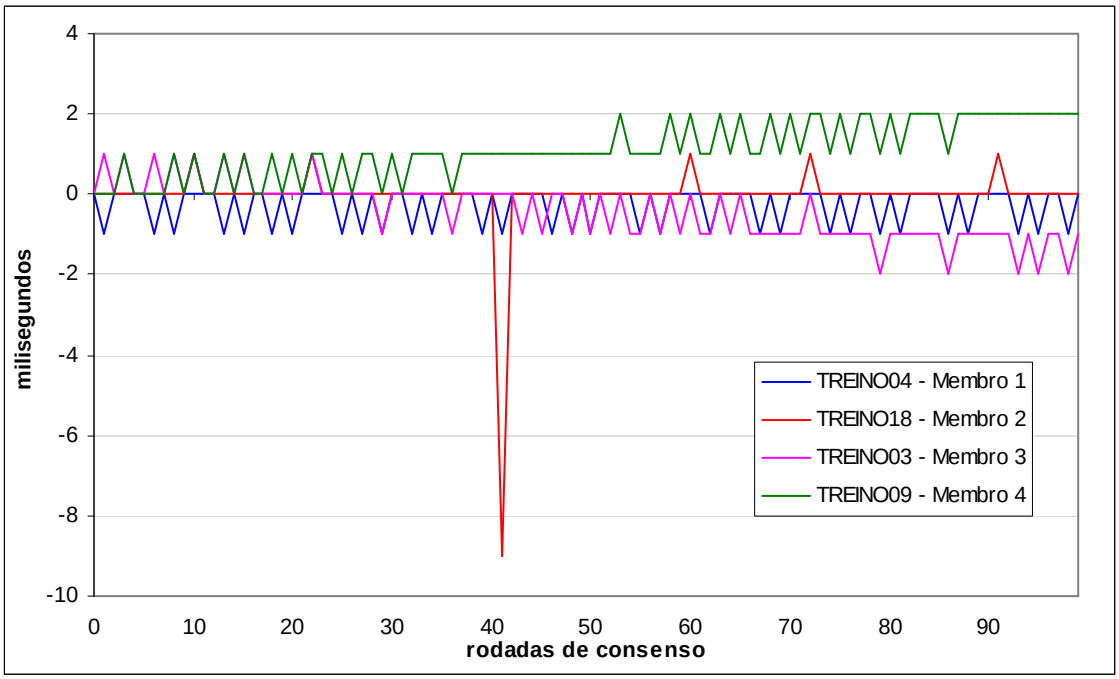

Gráfico 5.1 – Variação da desincronia durante um teste com 100 consensos envolvendo 5 membros.

Para minimizar o problema com a variação no cálculo da desincronia, fizemos uma pequena alteração no *DesincCalc* para que ele calculasse a desincronia entre a MR e os outros membros repetidamente até que encontrasse o mesmo valor nas últimas 5 rodadas de cálculo.

O nosso intuito ao alterar o *DesincCalc* foi o de garantir que o primeiro consenso não fosse calculado fora da média geral da desincronia. Um exemplo deste problema pode ser observado no Gráfico 5.1, notem que o valor calculado para a desincronia na rodada 42 estava muito diferente do calculado em todas as outras rodadas de consensos. Como a ocorrência deste problema foi registrada de forma isolada, repetindo o cálculo por 5 vezes, conseguimos minimizar os problemas decorrentes da falta de precisão no cálculo da desincronia.

A análise da variação da desincronia permitiu identificarmos posteriormente qual a sua influência no cálculo do desempenho das duas FCGs. Maiores detalhes sobre esta análise serão fornecidos na próxima seção.

Com a evolução dos testes verificamos que foi necessário armazenar dados sobre a utilização de memória por parte da aplicação de consensos. Utilizamos a instância da classe *Java.lang.Runtime* existente na *App_Consensus* para verificar

qual a quantidade de memória disponível para a JVM no momento dos testes. Através do método *java.lang.Runtime.freeMemory()*, verificamos a memória disponível em instantes diferentes.

Para registrar todos os tempos mencionados até aqui, criamos a classe *Log*. A aplicação de consenso possuía um atributo deste tipo para o qual delegava a responsabilidade de gravar as informações sobre os consensos em arquivo de log.

Nos testes realizados a classe *Log* mantinha uma estrutura de dados em memória e somente registrava as informações em disco após o término da bateria de testes.Os dados eram armazenados na memória RAM em um objeto do tipo *Vector* chamado de **`buffer_`**. Ao finalizar uma bateria de testes, a aplicação de consensos executava o método **`Log.flush()`** e o conteúdo do **`buffer_`** então era gravado no disco rígido.

Como o método **`Log.flush()`** apenas era executado após a bateria de testes ter sido concluída, as atividades executadas pelo sistema operacional para gravar os tempos de envio e recebimento das mensagens em disco não concorreram com o envio e recebimento de mensagens, evitando interferências nos resultados obtidos.

Após a execução de cada bateria de testes, a *App_Consensus* gerava um arquivo de log contendo a marca de tempo em que os seguintes fatos ocorreram:

- Mensagens recebidas notificando as alterações no grupo de membros;
- Mensagens recebidas do grupo com valores sugeridos por membros remotos;
- Mensagens enviadas ao grupo com valores sugeridos para uma rodada de consenso;
- Registro de obtenção de um novo consenso;
- Valores com a desincronia (somente no computador utilizado como referência para o cálculo da desincronia);
- Informações sobre o consumo de memória durante a execução das diversas rodadas de consensos.
- Informações sobre o tempo gasto para executar a coleta de lixo (a ser descrito adiante).

Com base nos dados armazenados nos arquivos de log, foi possível calcular os três tipos de métricas utilizados para determinar o tempo gasto na obtenção dos consensos.

5.3 Análise dos resultados.

Nesta seção apresentaremos os resultados dos testes com a aplicação de consensos, destacaremos como tratamos alguns fatores externos para minimizar seu impacto nos resultados e faremos uma análise visando identificar como os modelos de estruturação interna das FCGs influenciaram o desempenho dos testes.

Conforme descrito na seção 5.2, realizamos 3 grupos de testes. Os dois primeiros grupos serviram para identificarmos alguns fatores externos que estavam influenciando os resultados dos testes executados. A motivação para realizar o segundo e o terceiro grupo de testes foi o de eliminar / controlar os fatores que estavam dificultando a análise do desempenho das duas FCGs, sobre a perspectiva da sua forma de estruturação interna.

Faremos uma breve explanação sobre os problemas encontrados nos dois primeiros grupos de testes e em seguida apresentaremos os detalhes de como estes problemas foram tratados e quais os resultados obtidos no terceiro e último grupo de testes. Finalmente, na seção 5.3.2 apresentaremos um resumo de quais foram as principais contribuições desta pesquisa.

Nos dois primeiros grupos de testes identificamos a presença de alguns consensos com tempos para conclusão muito superior à media geral dos consensos. Concluímos que estas grandes variações estavam sendo provocadas por fatores alheios aos testes, que durante a realização de algumas rodadas de consensos estavam se manifestando.

A nossa abordagem foi tentar identificar e eliminar estes fatores. Quando não fosse possível elimina-los, tentamos contornar os seus efeitos indesejados, controlando a sua ocorrência. Segue abaixo um resumo dos principais fatores encontrados:

- **Falta de sincronia entre o envio e recebimento das mensagens entre os membros.** A própria natureza da aplicação de consenso introduz atrasos na obtenção dos consensos. Este fato ocorre, pois os membros

apenas enviam os valores sugeridos para uma nova rodada de consenso, após terem obtido o consenso da rodada vigente. Como, tipicamente, um dos membros obtem o consenso primeiro, ele então envia o seu valor sugerido para o novo consenso e precisa aguardar que os outros membros obtenham o valor consensual da rodada anterior para então enviar seus respectivos valores. Este fato introduz um atraso devido à falta de sincronia entre os membros no envio das mensagens com valores sugeridos. Enquanto o primeiro membro a enviar o valor sugerido para o consenso possui valores altos para a rodada, o último irá obter o valor consensual com tempos muito baixos, pois quando ele envia o seu valor para a rodada em questão, todos os valores sugeridos pelos membros remotos já foram recebidos, neste caso ele não precisa aguardar por nenhuma mensagem.

- **Erro de implementação no iBusTFE -** Registramos que durante os testes, quando utilizamos o iBusTFE, não houve crescimento das fábricas de objetos (vide seção 4.2.1) apenas quando o grupo tinha 3 membros. Verificamos que a explicação para este fato e que as mensagens remotas enviadas para a classe *Newtop* (vide seção 3.4.2) não estavam sendo devolvidas para suas fábricas, devido a um erro de programação no iBusTFE. Isto fazia com que as fábricas tivessem que crescer aumentando o número de objetos disponíveis.

  As fábricas de objetos, durantes os testes, estavam configuradas para serem iniciadas com 250 objetos. Nas baterias com 3 membros são recebidas 2 mensagens de membros remotos por rodada. Portanto, cada membro recebe 198 mensagens remotas. Desta forma, explica-se porque não foi registrado o aumento da fábrica de eventos nos testes realizados com 3 membros. Já em grupos de 4, 5 e 6 membros são recebidas por bateria de testes 297, 396 e 495 mensagens remotas, respectivamente. Assim, foi necessário que as fábricas de eventos crescessem ao menos uma vez durante os testes com estes tamanhos de grupo.

  Eliminamos o erro de programação antes de executar os testes finais. Este fato certamente teria graves conseqüências nos testes, pois a

operação de crescimento das fábricas de objetos é onerosa. Ela aloca o dobro do número de objetos da criação ou último aumento da fábrica na memória. No nosso caso, um aumento de 500 objetos.

- **Utilização de uma rede de comunicação assíncrona –** A rede de comunicação utilizada é assíncrona, assim não há a garantia de tempo para a entrega dos pacotes. Apesar de controlarmos todo o tráfego no momento dos testes, minimizando o problema de atrasos na entrega dos pacotes, ainda havia a possibilidade de que em um determinado momento vários membros enviassem mensagens concorrentemente causando atrasos na entrega das mensagens à aplicação.
- **Tendência a consensos com 50ms de duração –** Os tempos registrados com as duas FCGs para a realização de uma rodada de consenso tendiam a 50ms. Isto ocorria, pois ambas utilizam a classe *NEWTOP* para controlar a ordenação total. E na implementação desta classe foi criada uma Thread chamada de *Delivery* que verifica se existem blocos completos. Em caso positivo, esta Thread envia as respectivas mensagens para a classe *Stack* ou para a entidade *AppiBusTFE* para que elas fossem entregues à aplicação de consenso.

  Após verificar se existem mensagens a serem enviadas a Thread *Delivery* executava o método **`java.lang.Thread.currentThread.sleep()`** [SUN02], passando como parâmetro o valor 50. Isto fazia com que a Thread *Delivery* fosse bloqueada por 50ms.

  Assim, quando os membros enviavam e recebiam as mensagens de forma síncrona os valores registrados para a obtenção do valor consensual tendiam a ser muito próximos a 50ms. Isto ocorre porque enquanto a Thread *Delivery* permanecia bloqueada eram enviados para o grupo e recebidos os valores sugeridos para a rodada de consenso vigente. Ao ser acordada a Thread, verifica que todas as mensagens necessárias para a obtenção do consenso foram recebidas e então entrega as mensagens com blocos completos para a *App_Consensus*.

  Este fato estava mascarando os resultados, pois nos primeiros testes os tempos tendiam a 50ms, dificultando a análise dos tempos obtidos em

função do modelo de estruturação. Pois, não havia a garantia de que as mensagens seriam entregues à aplicação assim que fossem recebidas.

- **Coleta de lixo em momento não apropriado -** Suspeitamos que o GC estava sendo executado durante algumas rodadas de consensos e estava competindo com as *Threads* das FCGs pelos recursos do sistema operacional.

  Provavelmente o GC estaria afetando o desempenho das duas FCGs, porém no iBusTFE a sua interferência pareceu ser mais marcante devido ao padrão de comportamento identificado.

  Analisando a implementação das duas FCGs, identificamos que o iBusTFE devido às fábricas de objetos tinha a tendência de possuir mais objetos em memória, pois ao ser instanciado eram alocados 250 objetos para cada fábrica de eventos do tipo *AppEvent* e *EventAck*. Já no caso do iBusTF, os objetos eram alocados sob demanda e depois liberados no primeiro GC a ser executado, não sendo necessário manter um conjunto de objetos não utilizados na memória como no iBusTFE.

  Por outro lado, as fábricas de objetos beneficiaram o iBusTFE sempre que houve a necessidade de utilizar produzir novo evento (p.e.: no recebimento de envio de mensagens), pois não era necessário instanciar um novo objeto, bastava solicitar à fábrica um dos eventos previamente criados.

  Esta abordagem fez com que diversos objetos se mantivessem alcançáveis na memória, pois a fábrica mantinha uma referência para todos seus objetos. Desta forma, os objetos mantidos nas fábricas não seriam afetados pelo GC. Porém, como o iBusTFE utiliza a classe Newtop, houve a necessidade de criar objetos do tipo *MessageEvent* no recebimento e envio de mensagens ao grupo. Estes objetos estavam sujeitos a serem recolhidos pelo GC.

  Como não conseguimos desabilitar a coleta de lixo, optamos por controlar a sua execução para eliminar a interferência do GC nos consensos. Forçamos que o GC fosse executado sempre entre as rodadas de consensos, deste modo ele não concorreria com a FCG no momento do recebimento ou envio de alguma mensagem. Assim,

concluímos que seria necessário fazer alguns ajustes na *App_Consensus* e que deveria ser efetuado mais um grupo de testes.

O fato de ambas FCGs utilizarem protocolos de comunicação não confiáveis, não influenciou negativamente os testes, pois não registramos a ativação do mecanismo de recuperação de mensagens (vide seção 3.4.6.1) em nenhum teste com as FCGs. Assim, podemos concluir que durante os testes não houve a perda de pacotes na rede relativos a mensagens de consensos. Atribuímos este fato à infra-estrutura de comunicação utilizada, ou seja, uma rede local com o tráfego controlado, onde a taxa de perda de pacotes é baixa.

Apresentaremos na seção 5.3.1 algumas considerações sobre os testes realizados, os resultados obtidos e a análise comparativa dos dados das duas FCGs.

5.3.1 Grupo de testes final

Ao decidir executar o terceiro e último grupo de testes demos continuidade ao processo de eliminação dos fatores que estavam dificultando a nossa análise. Para demonstrar qual foi a abordagem adotada, organizaremos esta seção da seguinte forma: primeiro apresentaremos as alterações realizadas nas FCGs e na *App_Consensus* para controlar os fatores identificados. Em seguida, apresentaremos os dados obtidos nos testes e faremos algumas considerações sobre eles.

A seguir, elencamos os fatores que pretendíamos eliminar e quais as alterações realizadas:

5.3.1.1 Eliminar a desincronia existente no envio das mensagens com valores sugeridos.

No primeiro grupo de testes identificamos que a falta de sincronia entre os membros tinha sido o fator externo com maior influência nos resultados. Este fato ocorria, pois os membros tipicamente demoravam tempos diferentes para obter o consenso em uma mesma rodada, concluindo-o em momentos diferentes.

Para resolver o problema da falta de sincronia no envio das mensagens, alteramos a *App_Consensus* para que todas as rodadas de consensos fossem iniciadas em uma hora predeterminada. Ou seja, no início de cada bateria de

consensos, foram agendados com todos os membros do grupo os horários para o envio das mensagens com valores sugeridos para a primeira rodada de consenso. Assim, sincronizamos o início dos consensos nos membros do grupo. Para viabilizar o agendamento, foi necessário calcular a desincronia existente entre o membro 0 (máquina com o relógio de referência) e os outros membros do grupo (vide seção 5.2.2).

Antes do início da bateria de testes, o Membro 0 calculava a desincronia inicial existente entre ela e os outros membros do grupo. Em seguida, calculava o tempo previsto para o envio da mensagem com valor sugerido para o primeiro consenso de cada membro do grupo, levando em consideração o valor da desincronia calculado anteriormente. Então, o Membro 0 enviava uma mensagem para cada membro informando: (1) qual o horário calculado para o envio da primeira mensagem de consenso e (2) qual o intervalo para o envio das mensagens de consensos subseqüentes. Assim, cada membro ficava responsável por enviar as suas mensagens apenas nos momentos agendados.

Após a obtenção do consenso corrente, cada membro calculava qual o tempo que deveria “dormir” para então enviar a mensagem para o consenso subseqüente, tomando como base o valor do intervalo entre os consensos informado pelo Membro 0.

Para verificar a precisão do agendamento no envio das mensagens, passamos a medir em todas as baterias de testes realizadas no terceiro grupo, o tempo esperado para o envio das mensagens e o tempo efetivo de envio. Desta forma, foi possível calcular exatamente qual a desincronia existente entre os membros no envio das mensagens com valores consensuais sugeridos.

Constatamos que o agendamento reduziu muito o problema da desincronia entre os membros do grupo (vide seção 5.2.2). Porém, não conseguiu eliminá-lo por completo. Registramos variações no envio das mensagens com valores sugeridos na faixa de –10ms a 10ms. No Gráfico 5.2 podemos observar qual a variação entre o tempo esperado e o tempo efetivo de envio das mensagens com os valores sugeridos.

Nos Gráficos 5.2 e 5.3, apresentamos os valores registrados em 10 consensos que foram escolhidos aleatoriamente com o intuito de apresentarmos o erro no envio das mensagens consensuais.

Podemos observar que apesar do erro registrado em cada membro variar de –10ms a 10ms a desincronia entre os membros no envio das mensagens pode chegar a 20ms, nos casos onde um membro envia a sua mensagem antes do horário e outro depois, como, por exemplo, ocorreu no consenso 20 onde houve uma desincronia entre o membro “1” e o “2” de 12ms.

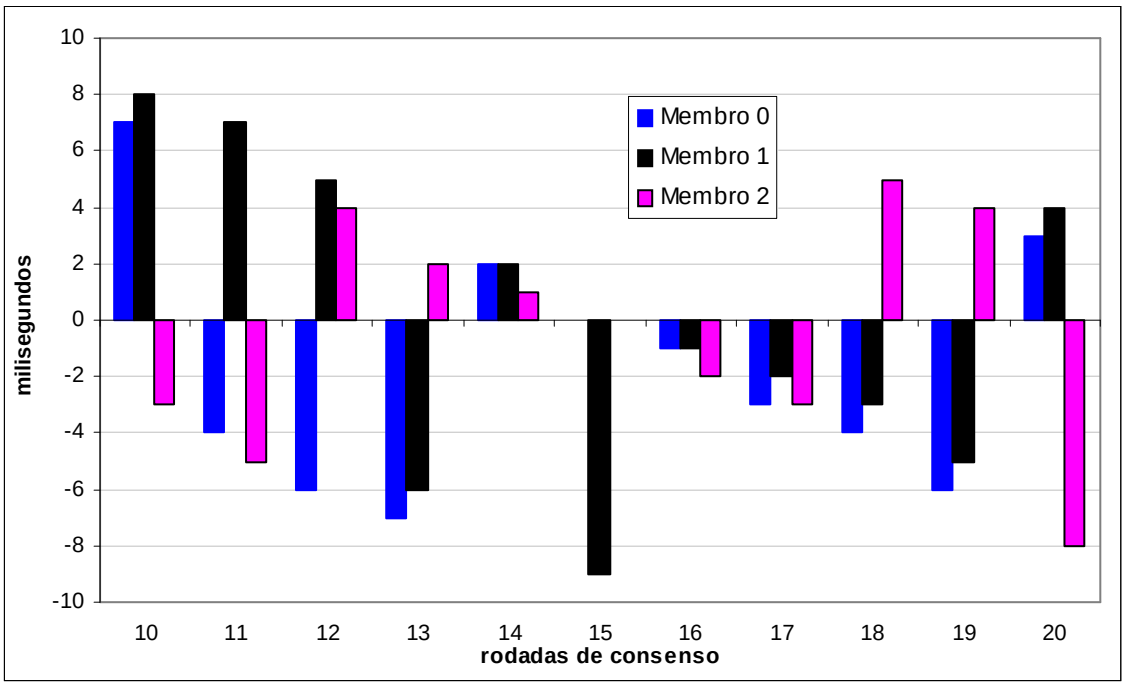

Gráfico 5.2– Erro no envio das mensagens de consensos registrada utilizando o iBusTFE com um grupo de 3 membros.

Por outro lado, caso todos os membros atrasassem ou o antecipassem o envio o impacto da desincronia seria minimizado, como ocorreu no consenso 17. Note que o membro “0” e o “2” enviaram suas mensagens 3ms após o horário previsto e o membro 1 também registrou um erro com 2ms além do tempo esperado. Como o referencial que nos interessa é a diferença entre os membros, neste caso não houve desincronia entre os membros “0” e “2” e a desincronia registrada com o membro 1 foi de apenas 1ms (vide Gráfico 5.3).

Portanto, para calcular o efeito dos atrasos / antecipações no envio das mensagens de início de consensos, tornou-se necessário medir o erro no envio das mensagens (tempo esperado – tempo efetivo) e comparar o erro registrado entre os

diferentes membros. Para isso, elegemos o tempo de envio das mensagens no Membro “0” como o referencial e calculamos o erro no envio das mensagens dos outros membros.

No Gráfico 5.3, exibimos os valores de desincronia registrados em 10 baterias de consensos, em um grupo contendo 3 membros, utilizando o iBusTFE. Os dados exibidos no Gráfico 5.3 foram obtidos a partir da mesma rodada de consenso expressa no Gráfico 5.2.

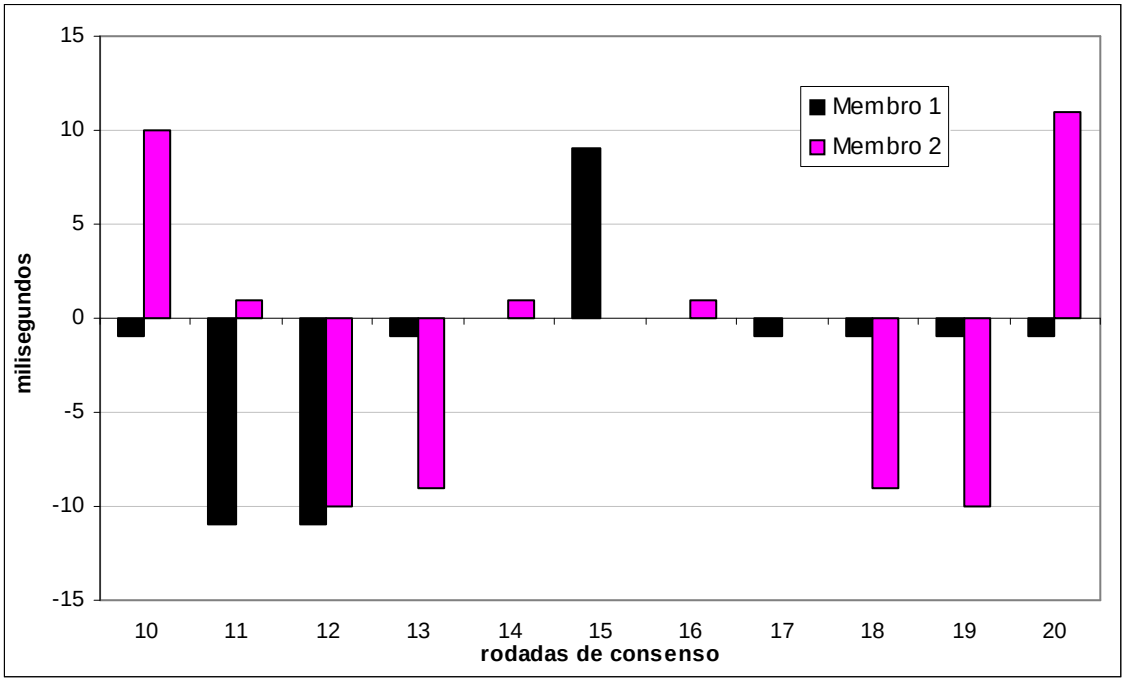

Gráfico 5.3 - Variação do erro em relação ao Membro 0 no envio das mensagens de consensos registrada utilizando o iBusTFE com um grupo de 3 membros.

Apesar de não eliminarmos o erro no envio das mensagens, com o agendamento das rodadas de consensos e o controle da desincronia existente entre os relógios das máquinas envolvidas nos testes foi possível mensurar com precisão o impacto da desincronia no tempo total para a obtenção dos consensos.

5.3.1.2 Evitar que o GC fosse executado durante algum consenso.

Um dos fatores que provavelmente estavam afetando os resultados dos dois primeiros grupos de testes foi a ocorrência de coletas de lixo durante os consensos.

Nossa suspeita era a de que os altos picos de valor registrados em ambas FCGs eram provenientes da ocorrência dos GCs [17].

Um objeto é considerado como lixo quando não existe nenhuma referência a ele na aplicação que está sendo executada. Uma implementação rudimentar do GC poderia simplesmente varrer todos os objetos alcançáveis pela aplicação e então eliminar os objetos que não foram alcançados existentes na memória dentro do contexto da JVM. Porém, este algoritmo demoraria um tempo proporcional à quantidade de objetos existentes na memória, o que prejudicaria as aplicações com grande número de objetos.

Para otimizar o desempenho, o GC faz algumas suposições sobre como a maior parte das aplicações utiliza os objetos. O funcionamento padrão do GC é baseado nestas suposições, porém é possível, através de parâmetros, ajustar o funcionamento do GC para casos particulares [SUN99]. A seguir explicaremos como conseguimos personalizar o GC para que o seu funcionamento se adequasse ao requerimento da *App_Consensus*.

A JVM 1.3.0, utilizada em todos os testes, incorpora 3 diferentes algoritmos para a coleta de lixo que são combinados utilizando a coleta de geração (*generational collection*). Este tipo de coleta presume que grande parte dos objetos possui um tempo de vida pequeno e apenas alguns poucos persistem alcançáveis após algumas iterações do GC.

O GC faz uso desta característica da maioria das aplicações para otimizar o processo de liberação de objetos. Para isso, ele gerencia a memória disponível divindindo-a em três gerações de objetos: (1) existe uma área para os objetos novos (também chamada de *éden*), (2) outra área para objetos antigos e (3) finalmente uma área para objetos que são utilizados por toda a execução e para objetos da própria JVM.

Todos os objetos da aplicação quando são criados são alocados no *éden* e a JVM supõe que a maioria dos objetos seja eliminada ali mesmo. Assim, quando o éden esgota a sua capacidade de armazenamento é executada uma coleta menor (minor collection). Este tipo de coleta copia os objetos ainda ativos do éden para a

[17] O mecanismo de GC é responsável por encapsular a complexidade da alocação / liberação de memória pelos objetos criados na aplicação.

área dos objetos antigos e então libera os objetos que permaneceram no éden. Quando a área dos objetos antigos é totalmente preenchida é executada uma coleta de lixo completa, que varre todos os objetos em memória eliminando os objetos não alcançáveis a partir da aplicação. Assim, a coleta de lixo completa é tipicamente mais lenta do que a coleta menor.

Observamos que a execução do GC está diretamente associada ao preenchimento das áreas de memória por ele gerenciadas. Portanto, o número de ocorrências do GC está inversamente associado à quantidade de memória alocada para a JVM. Quanto maior for a memória alocada, mais tempo demorará para preenchê-la e maior será o número de objetos que serão eliminados nas coletas de lixo menores, pois as coletas irão demorar mais a acontecer.

Por padrão a JVM aloca inicialmente 64Mb de memória para as aplicações. No decorrer da execução do programa, a cada coleta de lixo a JVM aumenta ou diminui a memória alocada, visando manter uma proporção de espaço livre em relação à quantidade de memória alocada pelos objetos ativos.

Portanto, se for possível alocar para a JVM uma quantidade inicial mínima de memória acima de 64Mb teremos dois benefícios: (1) haverá uma diminuição da incidência da coleta de lixo devido a um maior período entre duas coletas subseqüentes e (2) diminuirá a necessidade da JVM alocar / liberar memória para garantir o percentual de espaço livre.

Como o nosso intuito foi o de controlar a incidência da coleta de lixo, eliminando a sua ocorrência durante um consenso, adotamos algumas medidas para garantir que o GC fosse executado apenas no momento oportuno. Relacionamos abaixo quais as alterações que fizemos na *App_Consensus* com este intuito:

- Após a conclusão de cada rodada de consenso nos adotamos duas medidas: (1) inserimos um intervalo de 1000ms para iniciar o próximo consenso (vide agendamento de mensagens na seção 5.3.1.1) e (2) a *App_Consensus* passou a solicitar que o GC fosse executado após a obtenção de um consenso, utilizando o método `System.gc()`. Este método força a execução de uma coleta de lixo completa.

  Estas duas medidas combinadas permitiram que fosse executada uma coleta de lixo completa no intervalo entre duas rodadas de consensos subseqüentes. O nosso intuito foi o de liberar a memória no éden e na

geração de objetos antigos, para que o GC não fosse executado espontaneamente durante algum consenso.

Assim, após a obtenção de um valor consensual, os membros executavam uma coleta de lixo e então calculavam o tempo que deveriam "dormir" até iniciar o próximo consenso, determinando assim, quando enviar a mensagem com valor sugerido para o próximo consenso. Deste modo, cada membro poderia "dormir" mais ou menos tempo, em função do tempo gasto para efetuar a coleta de lixo.

- Passamos a utilizar duas opções não padronizadas da JVM para iniciar a *App_Consensus.* Este fato permitiu fazermos um ajuste fino do funcionamento do GC, alocando uma quantidade de memória inicial maior. Utilizamos a opção "–Xms180M" que determina que devem ser alocados inicialmente para a *App_Consensus* 180Mb de memória e a opção "–Xmx180M" que determina o máximo de 180Mb de memória alocada [SUN99]. Note que assim, determinamos que o valor inicial e o máximo de memória fossem iguais. Desta forma, garantimos que a JVM não iria aumentar ou diminuir a quantidade de memória alocada para a *App_Consensus* durante sua execução e minimizamos a probabilidade de ocorrência de execuções do GC durante algum consenso.
- Além disto, alteramos os tamanhos das fábricas de objetos do iBusTFE para 10 objetos. Como todos os objetos estavam sendo devolvidos para suas fábricas antes da *App_Consensus "*dormir" não era necessário possuir fábricas com 250 objetos, como vinha sendo utilizado nos grupos de testes anteriores. Esta medida visou diminuir o consumo de memória no iBusTFE para inibir que o GC fosse executado em um momento não apropriado.

Para verificar se com a adoção destas três medidas tínhamos conseguido controlar o momento em que foram executadas as coletas de lixo, utilizamos a versão 1.4.0 da JVM para fazer alguns testes adicionais. Esta versão da JVM possui uma opção não padrão para inicialização das aplicações (-Xloggc:[arquivo]) que permite criar um arquivo de log com as informações sobre todas as ocorrências do GC. Neste log é gerada uma linha para cada ocorrência do GC e entre outras informações existe o tipo da coleta executada (completa ou menor). Com estes

dados, foi possível determinar se o GC estava sendo executado durante algum consenso.

Fizemos testes com as duas FCGs com grupos de 5 membros e com baterias contendo 100 e 500 rodadas de consensos. Não fizemos testes com grupos menores, pois em grupos maiores a quantidade de memória alocada para armazenar as mensagens recebidas é superior do que em grupo menores. Assim, se o GC não se manifestasse em grupos grandes certamente ele não o faria em grupos menores.

Analisando o log gerado, constatamos que o número de coletas de lixo completas que foram executadas correspondia ao número de consensos. Ou seja, apenas estavam sendo executadas as coletas comandadas explicitamente pela *App_Consensus.* Registramos a ocorrência de algumas coletas menores após a execução das coletas completas previstas, mas estas coletas não têm nenhum efeito sobre os testes, pois estavam sendo executadas após a realização das rodadas de consensos previstas, não influindo nos tempos dos consensos.

5.3.1.3 Eliminar o tempo de espera para a entrega dos blocos completos à aplicação que fazia com que os consensos tendessem a 50ms.

Identificamos no primeiro grupo de testes que os tempos dos consensos tendiam a ser executados em 50ms, este fato ocorria em decorrência da Thread *Delivery*, utilizada na classe *NEWTOP*.

Esta Thread é responsável por entregar todas as mensagens relativas aos blocos completos à aplicação. Após uma entrega, ela “dormia” por 50ms para então verificar se havia novos blocos a serem enviados para a aplicação.

Esta abordagem influenciava os tempos dos consensos, retardando-os e não permitindo medirmos o funcionamento das duas FCGs a plena carga, situação onde poderíamos analisar o impacto das estruturais das FCGs nos tempos.

Para eliminar esta interferência, fizemos uma alteração na Thread *Delivery* eliminando a linha de código que fazia com que a Thread em questão dormisse por 50ms após entregar os blocos completos à aplicação.

Após executar alguns testes, verificamos que as duas FCGs tiveram seus tempos reduzidos drasticamente, pois a Thread *Delivery* passou a verificar ininterruptamente se havia blocos completos.

5.3.1.4 Tornar a análise sobre os dados obtidos mais rica utilizando novas formas para metrificar o desempenho das FCGs.

Ao efetuar as alterações necessárias nas FCGs e na *App_Consensus* para executar o terceiro grupo de testes, identificamos novos indicadores que seriam úteis na análise estrutural das duas FCGs. Assim, decidimos manter as métricas utilizadas nos dois primeiros grupos e adicionalmente passamos a medir / calcular novos indicadores. Abaixo relacionamos quais novos indicadores que utilizamos no terceiro grupo de testes:

- Começamos a utilizar mais duas formas de mensurar os tempos de consensos. Para isso, foi necessário calcular a desincronia existente entre os relógios dos membros que participaram dos testes (vide seção 5.2.2).
- Passamos a registrar as informações sobre o consumo de memória em cada consenso, ou seja, medimos qual a quantidade de memória disponível antes e depois de executar o GC. Desta forma, foi possível saber qual a quantidade de memória consumida em cada consenso e qual a quantidade liberada ao executar o GC. Registramos também o tempo gasto para efetuar a coleta de lixo.
- Medimos qual o erro no envio das mensagens com valores sugeridos para os consensos e qual a desincronia existente entre os membros (vide seção 5.3.1.1).

Com estes novos dados, pretendemos conseguir analisar sobre diferentes aspectos o funcionamento das duas FCGs. Ressaltamos, que cada uma das métricas utilizadas oferece uma perspectiva diferente para a análise.

Após efetuar as alterações necessárias para a adequação das FCGs / *App_Consensus* e verificar a sua eficácia, executamos 3 baterias de consensos contendo 500 rodadas cada com grupos de 3 membros, visando determinar qual o comportamento das duas FCGs.

Nos Gráficos 5.4 e 5.5 estão expressos os tempos para a obtenção de 500 rodadas de consensos utilizando o iBusTFE e o iBusTF, respectivamente. Notamos que as alterações efetuadas nas FCGs e na *App_Consensus* surtiram efeitos expressivos.

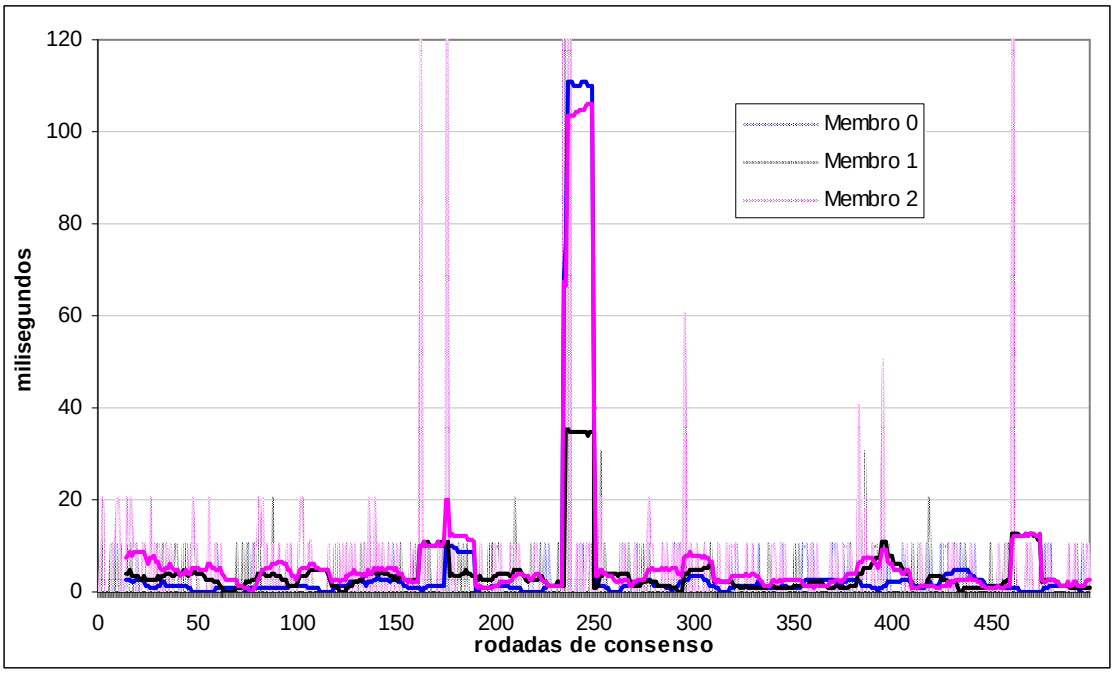

Gráfico 5.4 - Tempos para obtenção dos consensos em uma baterias de testes contendo 500 rodadas de consensos com um grupo de 3 membros, utilizando o iBusTFE.

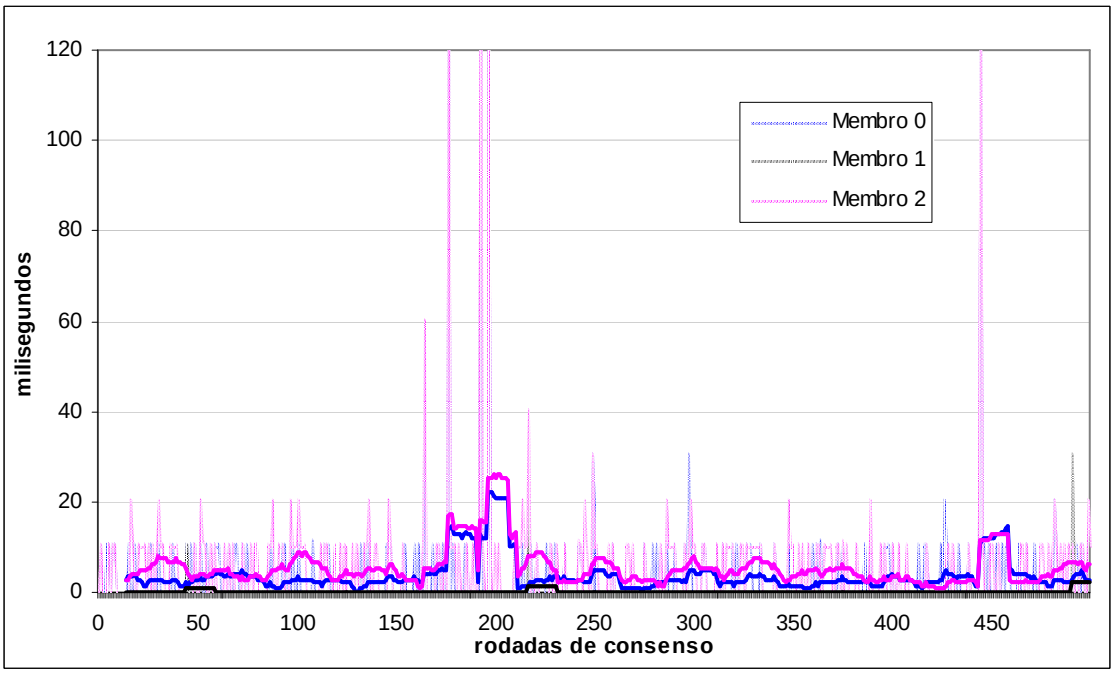

Gráfico 5.5 - Tempos para obtenção dos consensos em uma baterias de testes contendo 500 rodadas de consensos com um grupo de 3 membros, utilizando o iBusTF.

Em ambas as FCGs, o tempo médio para obtenção dos consensos que estava variando em torno de 80-90ms, antes de efetuarmos as alterações apresentadas nesta seção, ficou abaixo de 10ms no terceiro grupo de testes. O motivo principal para esta redução foi a alteração efetuada na Thread *Delivery* (vide seção 5.3.1.3).

Outra alteração de comportamento importante foi a diminuição expressiva de picos de consensos com alto valor. Ainda foram registrados alguns consensos com tempos muito acima da média geral, porém este número foi minimizado e não identificamos mais um padrão de comportamento repetido ao longo do tempo como ocorria nos grupos de testes anteriores. Acreditamos que a causa para a estabilização das FCGs tenha sido o fato de termos controlado a incidência do GC durante os consensos.

Nestas três baterias de testes, as duas FCGs tiveram um comportamento similar, porém o iBusTF se mostrou mais estável no que diz respeito aos consensos com alto tempo. O consenso mais demorado registrado com o iBusTF demorou 180ms e foram registrados em uma bateria de 500 consensos apenas 8 consensos com tempos superiores a 100ms, totalizando 1244ms. No iBusTFE o consenso mais demorado durou 1001ms e foram registrados 12 consensos acima de 100ms que totalizaram 4497ms.

Não conseguimos identificar, precisamente, qual a causa dos picos de consensos com alto valor que foram registrados. Acreditamos que um dos fatores que estejam provocando estes picos seja o crescimento de alguma estrutura de dados residente em memória, como vetores e hashtables, durante a realização de algum consenso.

Cabe ressaltar, que todas as mensagens enviadas e recebidas do grupo foram armazenadas em memória de forma cumulativa, não sendo liberada a memória até o final da bateria de consensos. Como utilizamos um *epochSize* com valor equivalente a 1000 e fizemos baterias de 500 e 100 consensos, as mensagens apenas seriam liberadas após a conclusão da primeira época, ou seja, após o consenso de número 1000.

Por conseguinte, o consumo de memória registrado utilizando o iBusTFE e o iBusTF foi progressivo, como pode ser visto nos Gráficos 5.6 e 5.7, onde está expressa a quantidade de memória disponível para a JVM ao longo dos consensos.

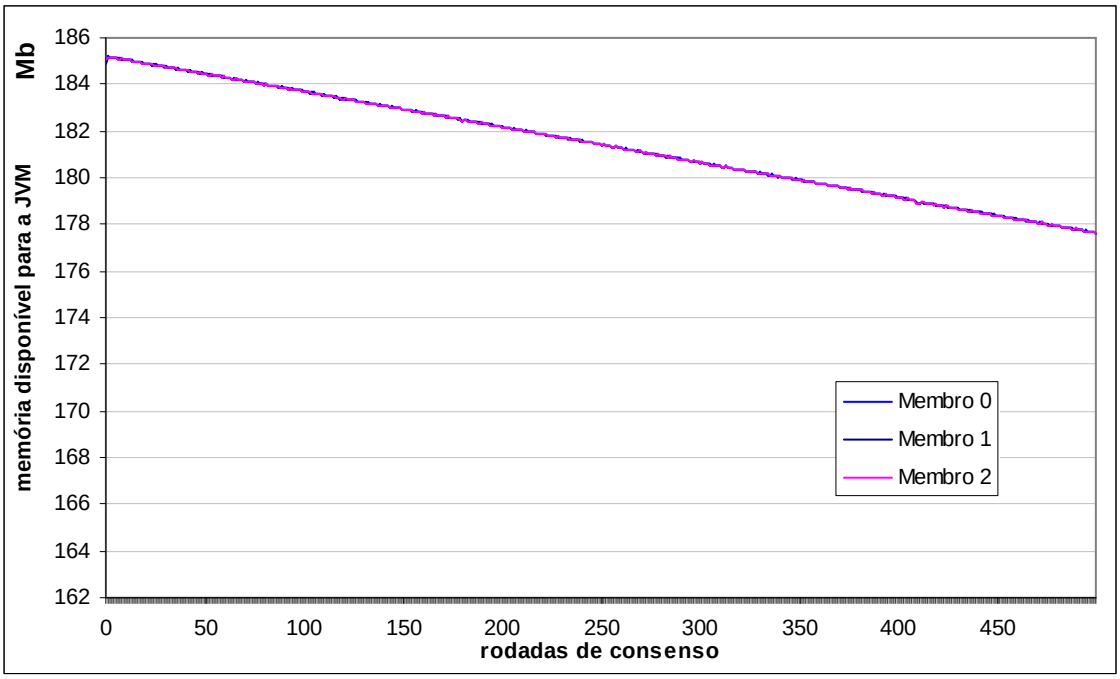

Gráfico 5.6 – Evolução da quantidade de memória disponível para a JVM durante as 500 rodadas de consenso, utilizando o iBusTFE com um grupo de 3 membros.

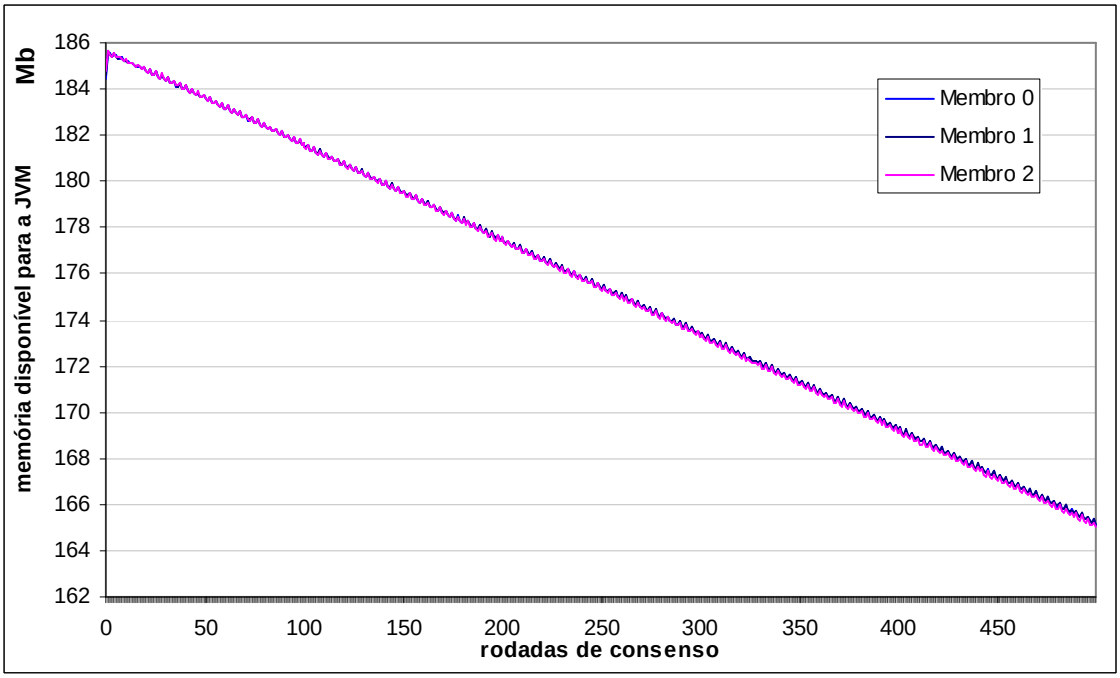

Gráfico 5.7 – Evolução da quantidade de memória disponível para a JVM durante as 500 rodadas de consenso, utilizando o iBusTF com um grupo de 3 membros.

Podemos observar que a memória disponível para as duas FCGs no primeiro consenso é de aproximadamente 185Mb e que ao longo das baterias de consensos a quantidade de memória disponível vai diminuindo sistematicamente. É importante lembrar que a cada rodada de consenso é executada uma coleta de lixo completa, portanto o consumo de memória progressivo se deve ao fato do tamanho da época utilizada não permitir que seja liberada a memória que está armazenando as mensagens recebidas e enviadas relativas à época atual.

Porém, acreditamos que haja outros fatores não identificados, além do aumento das estruturas de dados em memória, influenciando algumas rodadas de consensos inserindo os atrasos citados.

Registramos, que o iBusTF retem mais memória ao longo dos consensos do que o iBusTFE, conforme se pode observar nos Gráficos 5.6 e 5.7. Verificamos que o consumo registrado pelas duas FCGs não causou a execução de nenhuma ocorrência do GC não comandada pela *App_Consensus*.

Outro fato registrado referente à gerencia de memória, foi o de que o iBusTF consome e libera mais memória a cada coleta de lixo quando comparado ao iBusTFE. Calculamos a média aritmética de 10 baterias contendo 100 rodadas de consensos cada, utilizando um grupo com 3 membros, e verificamos que a média de memória liberada em cada ocorrência do GC foi de 215Kb e de 98Kb pelo iBusTF e pelo iBusTFE, respectivamente. Este fato é justificado pela utilização das fábricas de objetos no iBusTFE.

Aliado a este fato, verificamos que apesar de liberar menos memória, as coletas de lixo realizadas pelo iBusTFE demoram mais tempo, conforme pode ser observado nos Gráficos 5.8 e 5.9. Note que o iBusTF registra nas baterias finais tempos em torno de 150ms e o iBusTFE tempos em torno de 200ms, este fato nos levou a crer que o número de objetos que permanecem ativos no iBusTFE é maior do que o existente no iBusTF (outra característica do uso das fábricas de objetos). Este fato prejudica o desempenho do GC no iBusTFE, porém este fato não possui nenhuma relevância nos testes, já que as coletas de lixo foram executadas entre os consensos.

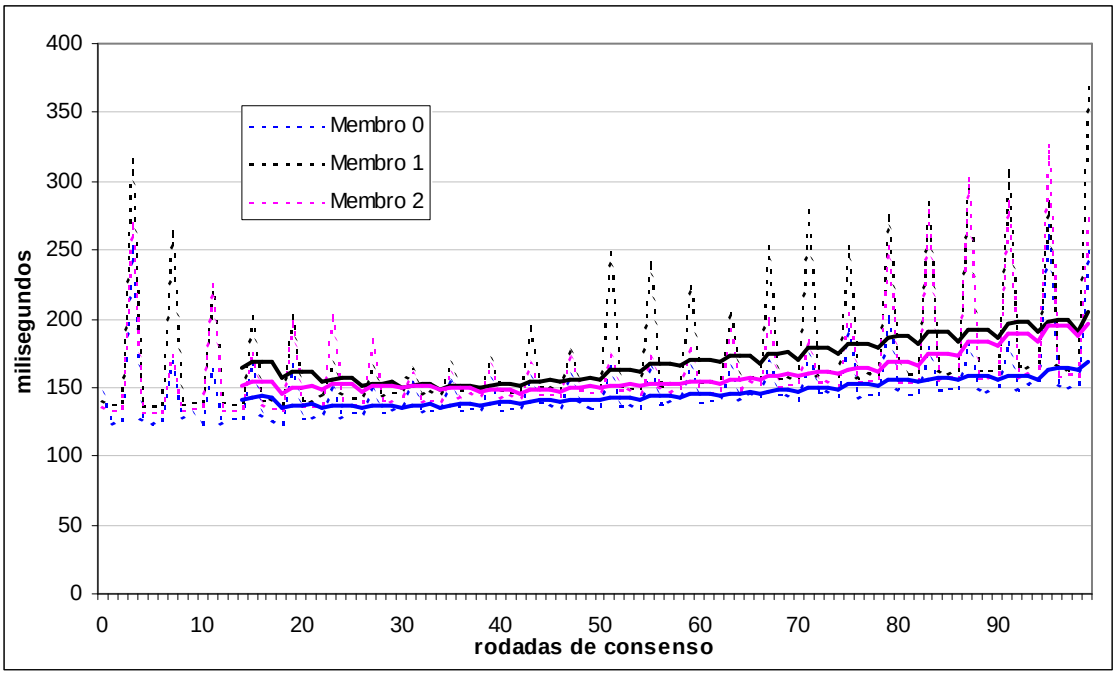

Gráfico 5.8 – Evolução dos tempos registrados para realizar as coletas de lixo completas, utilizando o iBusTFE com um grupo de 3 membros em 10 baterias com 100 rodadas de consenso.

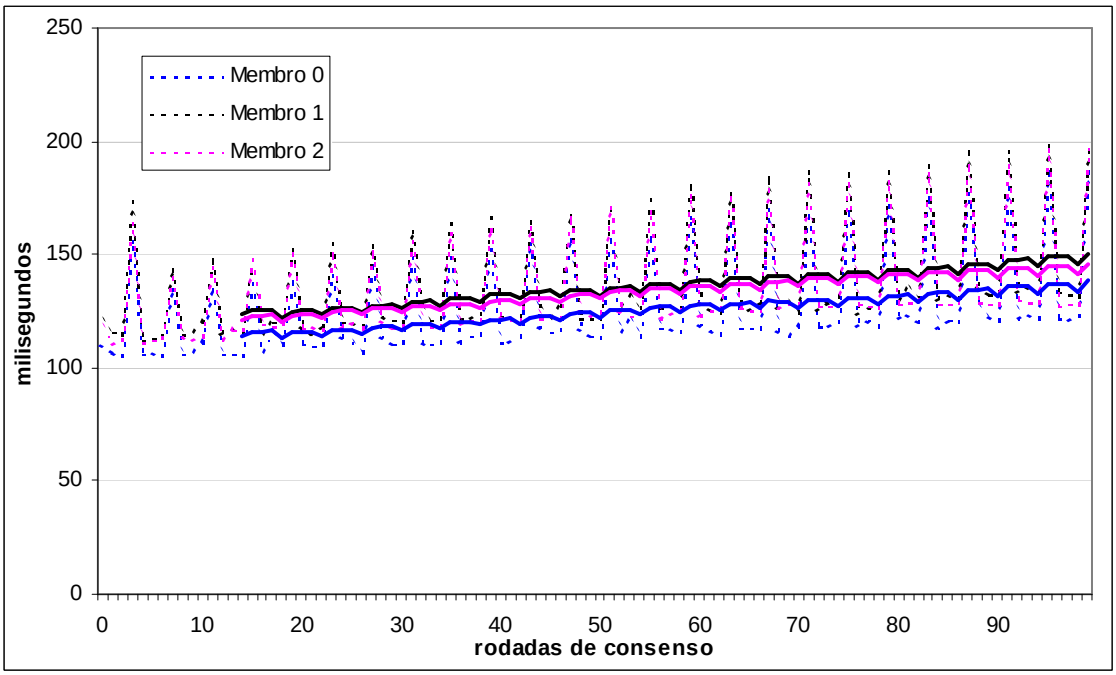

Gráfico 5.9 – Evolução dos tempos registrados para realizar as coletas de lixo completas, utilizando o iBusTF com um grupo de 3 membros em 10 baterias com 100 rodadas de consenso.

Nos testes preliminares com as baterias de 500 rodadas de consensos, observamos que a maior incidência de picos de consensos com altos valores surgiram apenas após o consenso 150, com grupos de 3 membros. Assim, decidimos trabalhar com baterias de consensos contendo 100 rodadas para poder minimizar os efeitos dos consensos de alto valor na nossa análise final.

Sendo assim, executamos 10 baterias de consensos, contendo 100 rodadas de consensos cada, com grupos de 3, 4 e 5 membros utilizando a *App_Consensus* com o iBusTFE e com o iBusTF.

Conforme esperávamos, constatamos que com os testes contendo 100 rodadas de consensos houve uma sensível redução em relação à incidência de consensos de alto valor. Com o iBusTFE não registramos nenhum consenso acima de 40ms nas 1000 rodadas de consensos (executadas em 10 baterias) e apenas 17 consensos tiveram seus tempos superiores a 20ms totalizando 494ms. Os tempos registrados com o iBusTF também foram considerados estáveis, apesar de termos registrado alguns picos de alto valor. Houve um consenso que demorou 1011ms e registramos 40 consensos acima de 20ms totalizando 3844ms [18].

Note que após controlar o momento em que ocorre o GC, as duas FCGs passaram a ter o mesmo comportamento no que se refere a picos de alto valor. A cada bateria de testes, registramos consensos de alto valor com ambas as FCGs. Porém, a incidência destes consensos, foi drasticamente reduzida como pode ser observado nos dados expressos no parágrafo anterior.

Nos Gráficos 5.10 e 5.11, apresentamos os tempos médios registrados para a obtenção das 100 rodadas de consensos com grupos de 3 membros. Os dados representados nestes dois Gráficos foram obtidos considerando os tempos de início e fim dos consensos medidos a partir do relógio local de cada máquina.

Podemos notar que, o comportamento das duas FCGs é muito similar caso não consideremos os picos de alto valor registrados no iBusTF. Observamos que o tempo médio registrado em todos os membros utilizando as duas FCGs ficou abaixo de 5ms. Fica claro que os picos tiveram uma maior influência sobre os consensos realizados com o iBusTF.

---

[18] Tempos calculados tomando como base que o início e o fim do consenso foram medidos no mesmo relógio (vide seção 5.2.2).

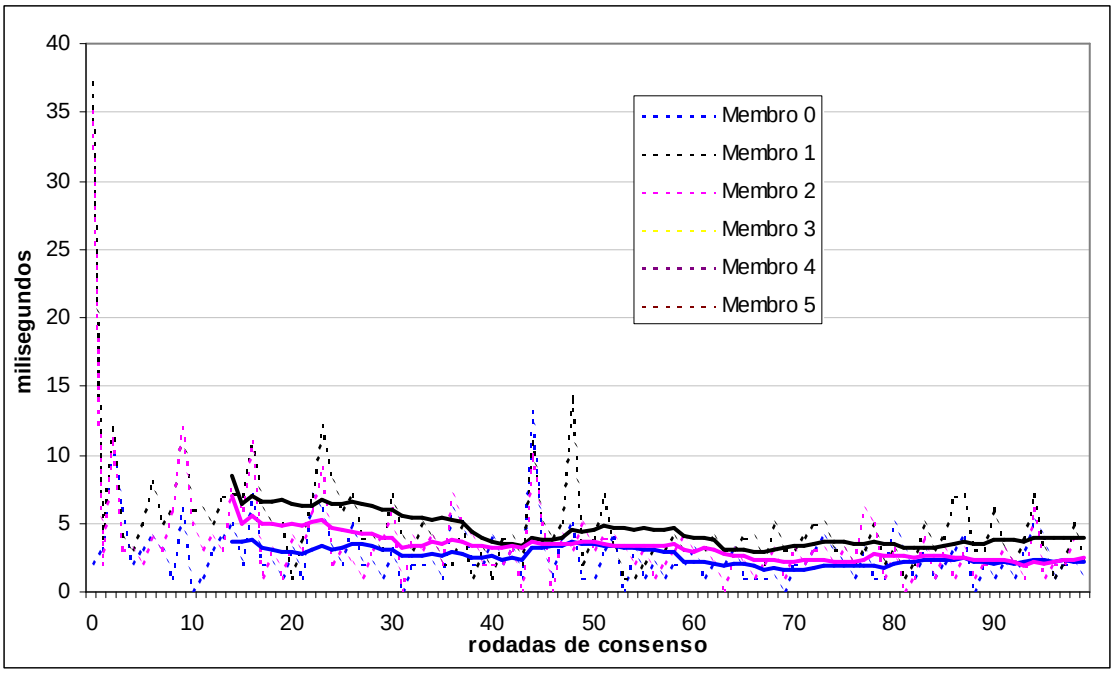

Gráfico 5.10 - Tempos médios para obtenção dos consensos em baterias de testes contendo 100 rodadas de consensos com um grupo de 3 membros, utilizando o iBusTFE.

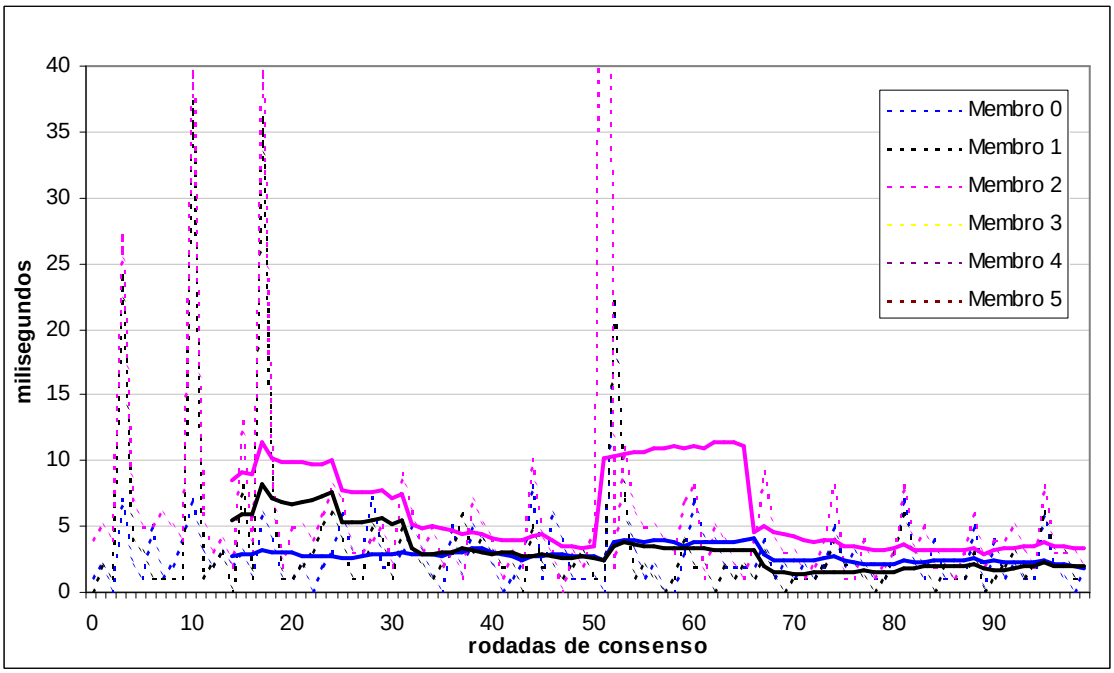

Gráfico 5.11 - Tempos médios para obtenção dos consensos em baterias de testes contendo 100 rodadas de consensos com um grupo de 3 membros, utilizando o iBusTF.

Com o intuito de diminuir o impacto dos consensos de alto valor nos dados obtidos, analisamos os resultados utilizando outras duas métricas: (1) o menor e (2) o maior tempo que qualquer um dos membros demorou para obter um consenso. Neste tipo de medição consideramos que um consenso foi iniciado quando o primeiro membro enviar sua mensagem com valor sugerido.

Os Gráficos 5.12 e 5.13, apresentam os valores obtidos pelo iBusTFE e iBusTF, respectivamente. Podemos observar que os picos de alto valor podem ser identificados nos tempos registrados utilizando o iBusTF, porém o seu impacto sobre a média é minimizado, em função da métrica aplicada.

Este fato permitiu ratificarmos a semelhança, observada anteriormente, no desempenho obtido pelas duas FCGs. Obtivemos tempos médios de 2ms e 3ms para o tempo mínimo e 12ms e 10ms para o tempo máximo, utilizando o iBusTFE e o iBusTF, respectivamente (vide Tabela 5.4).

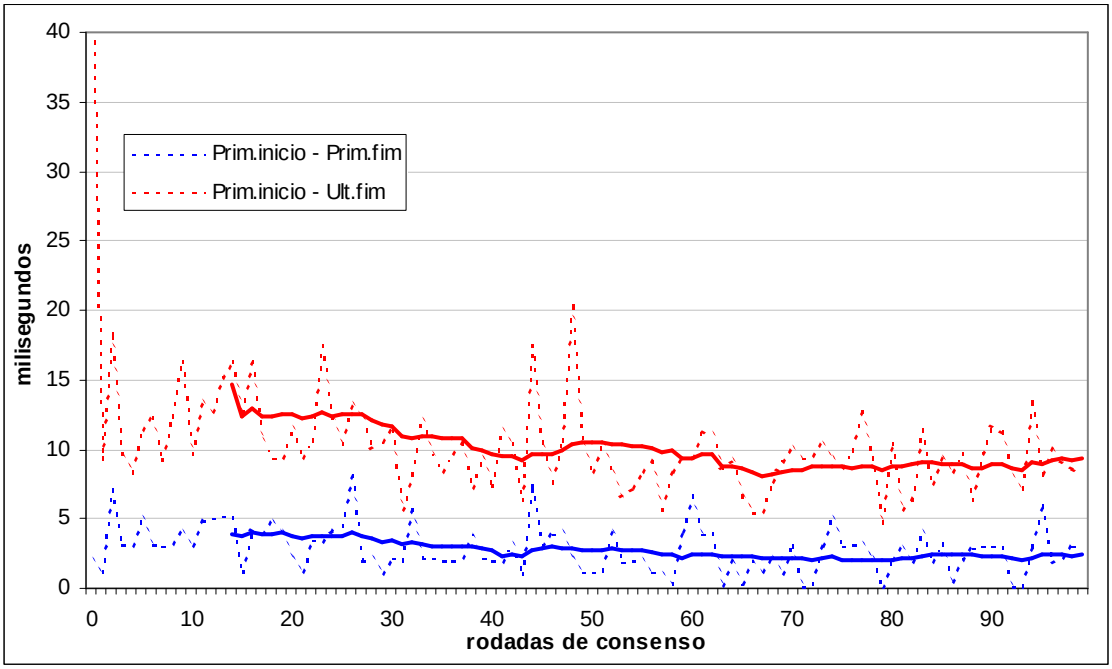

Gráfico 5.12 – Tempos médios mínimo e máximo registrados para obtenção de 100 rodadas de consensos, com um grupo de 3 membros, utilizando o iBusTFE.

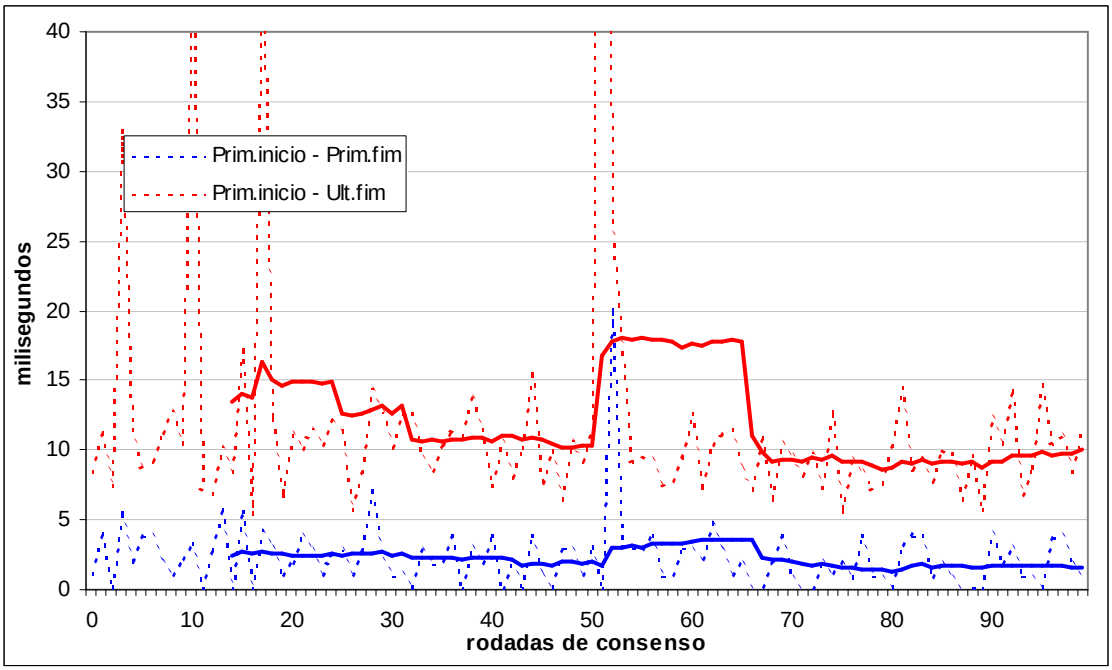

Gráfico 5.13 - Tempos médios mínimo e máximo registrados para obtenção de 100 rodadas de consensos, com um grupo de 3 membros, utilizando o iBusTF.

Se considerarmos a métrica utilizada onde o início e o fim dos consensos são medidos no relógio da máquina local os dados obtidos com as duas FCGs também são muito parecidos. Obtivemos utilizando o iBusTFE médias de 3ms, 5ms e 4ms e com o iBusTF médias de 3ms, 3ms e 6ms para os membros 0, 1 e 2, respectivamente.

Na Tabela 5.4, estão expressos os tempos médios mínimos e máximos registrados no terceiro grupo de testes. Observem que com grupos de 3, 4 e 5 membros e baterias de testes com 100 rodadas de consenso, os tempos registrados por ambas as FCGs foram muito similares.

Tamanho do grupo	iBusTFE		IBusTF	
	Mínimo	Máximo	Mínimo	Máximo
3 membros	3ms	10ms	2ms	12ms
4 membros	4ms	16ms	4ms	14ms
5 membros	6ms	27ms	4ms	20ms

Tabela 5.4 – Tempos médios mínimo e máximo registrados para obtenção de consensos com baterias de testes contendo 100 rodadas de consensos.

Após realizar uma análise dos tempos para a obtenção dos consensos, por ambas FCGs e concluir que tiveram desempenho equivalente, decidimos analisar os tempos gastos para a execução das funcionalidades associadas às camadas no iBusTF e aos componentes no iBusTFE. O nosso intuito foi de efetuar uma análise comparativa entre os tempos das duas FCGs para estabelecer se houve ganhos e / ou perdas em decorrência do modelo de estruturação interna.

Na análise teórica dos modelos de estruturação para protocolos de comunicação, apresentada no capítulo 2, a expectativa era de que os resultados do iBusTFE fossem superiores aos registrados pelo iBusTF, em função da forma de estruturação interna das duas FCGs.

O uso do paralelismo seria uma das grandes vantagens do modelo baseado em canais de eventos. Por exemplo, no iBusTF para que uma mensagem seja enviada ou recebida do grupo é necessário que esta seja manipulada seqüencialmente pelas 6 camadas do iBusTF. Esta característica é imposta pelo modelo em camadas. No iBusTFE os eventos do tipo *appEvent* percorrem uma espécie de caminho crítico, ou seja, eles são manipulados de forma seqüencial apenas quando é estritamente necessário. Sempre que possível faz-se o uso do tratamento em paralelo de um mesmo evento. Assim, no envio de uma mensagem ao grupo, um *appEvent* é manipulado por 5 entidades de forma seqüencial e no recebimento de *appEvent* remotos apenas 4 entidades manipulam a mensagem de forma seqüencial.

Uma das situações em que o iBusTFE pode se beneficiar com o paralelismo ocorre quando mensagens do tipo *appEvent* são recebidas de membros remotos e devem ser tratadas pela componente NAK. Neste caso, a mensagem é recebida pelo componente *compGroupReceiver* e enviada para os componentes *compFifo* e *compNAK*. Estes dois componentes enviam o evento recebido para as entidades *AckSender* e *Orderer,* respectivamente*.* O tratamento destas mensagens pelas entidades é efetuado de forma paralela. Desta forma, a mensagem a ser enviada pela aplicação será encaminhada ao *compTF* assim que a entidade *Orderer* tratá-la. Não existe a necessidade de aguardar que o *AckSender* trate a mensagem recebida para poder envia-la ao *compTF* e, conseqüentemente, à aplicação de consensos. No iBusTF todas as mensagens recebidas apenas são enviadas para a aplicação de consensos após terem sido tratadas pela camada NAK.

Para aferir se foi registrado algum ganho em função do paralelismo existente no modelo baseado em canais de eventos e para conseguir maiores detalhes sobre o desempenho das duas FCGs, utilizamos a ferramenta de perfilamento chamada de Quantify [RAT02] para obter os seguintes dados:

- Relação completa de todos os métodos executados, com os tempos totais consumidos pelo próprio método e pelos métodos que ele chamou, número de vezes que os métodos foram executados, métodos que chamaram o método analisado e os que foram chamados por ele.
- Informações sobre cada linha de código executada, contendo o tempo total consumido por linha de código e pelos métodos invocados a partir dela.

Para obter os dados requeridos o Quantify interage com a JVM através da interface JVMPI (Java Virtual Machine Profiling Interface), que está descrita no padrão Java 2 [SUN02]. Como utilizamos a JVM 1.3.0 da própria Sun não tivemos problemas de compatibilidade entre o Quantify e a JVM utilizada.

Notamos que os tempos obtidos nos testes com o Quantify foram mais altos utilizando as duas FCGs do que os registrados apenas com as FCGs. Este fato já era esperado, pois o próprio Quantify concorre com as FCGs pelos recursos do sistema[19] .

Para focar a nossa análise nas diferenças estruturais das duas FCGs, estudamos quais os tempos gastos desde o recebimento de uma mensagem pela camada IPMCAST até a sua entrega para a *App_Consensus*. Obtivemos qual o tempo total gasto para que as classes inerentes a cada camada processassem suas mensagens e dividimos pelo total de mensagens processadas, assim obtivemos o tempo médio gasto por cada camada para processar uma única mensagem remota recebida. Por exemplo, identificamos que foram gastos 1251,9ms, para que as classes inerentes à camada TF no iBusTFE processassem 203[20] mensagens recebidas dos membros remotos. Logo, concluímos que para processar uma mensagem foram gastos em média 6,17ms.

---

[19] Para minimizar este tipo de problema o Quantify armazena as informações coletadas durante os testes em memória e apenas os grava no disco após a conclusão do teste.

[20] Registramos que foram recebidas 200 mensagens do tipo *AppEvent* dos membros remotos e 3 mensagens de inicialização.

Na Tabela 5.5, apresentamos qual o tempo médio gasto pelas FCGs para tratar uma mensagem remota recebida de algum membro do grupo. Dividimos os tempos em função das camadas do iBusTF. Isto foi possível, pois há um componente no iBusTFE para cada camada do iBusTF, com exceção da camada FRAG.

Camada / Componente	iBusTFE (ms)	iBusTF (ms)
IPMCAST	7,95	6,51
REACH	-	0,57
NAK	-	1,67
FIFO	0,72	0,40
FRAG	-	0,28
TF	6,17	0,39
STACK	0,09	2,90
Totais	14,92	12,71

Tabela 5.5 – Caminho crítico seguido por uma mensagem remota nas FCGs.

É importante ressaltar que somente estão expressos na Tabela 5.5, na coluna referente ao iBusTFE, os valores que fazem parte do caminho crítico. Ou seja, não apresentamos os valores registrados para a camada NAK, pois esta irá tratar a mensagem processada pelas entidades do componente NAK paralelamente ao envio da mensagem para a aplicação. Também não exibimos os valores para a camada REACH, pois ela não necessita tratar as mensagens do tipo *AppEvent* remotas e, finalmente, não existe tempo para a camada FRAG, pois ela não existe no iBusTFE[21].

Analisando os dados da Tabela 5.5, podemos notar que as duas FCGs apresentaram resultados gerais muito parecidos registrando uma diferença em torno de 2ms sobre o tempo total para o recebimento de uma mensagem remota. Proporcionalmente, os resultados obtidos com o Quantify são similares aos observados no terceiro grupo de testes. Porém, nos chamou a atenção a diferença

[21] As funcionalidades da camada FRAG foram incorporadas a partir do framework EVA (vide seção 4.2.1).

existente entre o tempo gasto para processar as mensagens relativas às camadas TF e STACK nas duas FCGs.

Em relação à camada STACK, identificamos que o iBusTF na classe *Stack* utiliza um objeto do tipo *ObjectInputStream* para recuperar / encapsular as informações enviadas a / de membros remotos. Este tipo de objeto é utilizado para recuperar dados serializados anteriormente. No nosso caso, os dados foram serializados para serem remetidos pela rede de comunicação e devem ser deserializados para serem acessados. Estas operações são realizadas utilizando os métodos `writeObject()` e `readObject()`, `respectivamente.` Estas operações são consideradas “pesadas”, pois impõem um alto custo em relação ao desempenho. Verificamos que somente para executar o método `readObject()`, o iBusTF gastou em média 2,48ms por mensagem remota recebida. No iBusTFE, todas as operações que envolvem serialização e deserialização de dados são efetuados pelo próprio framework EVA, e os tempos foram computados na Tabela 5.5 junto com os da camada IPMCAST.

Este detalhe de implementação fez com que os tempos registrados na camada STACK com o iBusTF fossem muito superiores aos do iBusTFE e que os da camada IPMCAST fossem inferiores. Se somarmos os 2,48ms gastos no iBusTF pela camada STACK para recuperar / encapsular os dados ao tempo gasto pela camada IPMCAST, teremos um tempo total de 8,99ms que é valor aproximado aos 7,95ms gastos pelo iBusTFE com operações similares.

Para averiguar qual a causa do baixo desempenho do iBusTFE, no que diz respeito à camada TF, utilizamos o Quantify e fizemos uma análise comparativa de quais as atividades foram executadas e os tempos registrados pelas duas FCGs na camada TF. Relacionamos abaixo os dois fatores que identificamos como as causas do baixo desempenho do iBusTFE no que se refere ao componente TF:

- O iBusTFE, através da entidade *TotalOrderer,* utiliza a classe *Newtop* para garantir a ordenação total. Utilizamos esta classe com o intuito de permitir que as duas FCGs tivessem implementações similares, facilitando a análise dos resultados.

  Para viabilizar o uso da classe *Newtop,* implementamos métodos para converter os eventos do iBusTFE para o iBusTF quando do recebimento de mensagens e vice-versa quando fossem enviadas

mensagens ao grupo. Verificamos que esta abordagem inseriu operações que trouxeram conseqüências negativas para o desempenho do iBusTFE.

A cada *AppEvent* remoto recebido foi necessário instanciar um objeto do tipo *iBus.MessageEvent,* transferir os dados contidos na mensagem recebida e enviar o evento criado para a classe *Newtop*. Registramos que em uma bateria de consensos com 100 rodadas foram gastos 109ms para efetuar estas operações. A cada mensagem enviada para o grupo também foi necessário criar objetos do tipo *iBus.MessageEvent* e os enviar para a classe *Newtop.* Verificamos que para 100 rodadas de consensos foram gastos 53ms com esta operação.

Quando os blocos se tornam completos a Thread *Delivery* os envia para a aplicação. Novamente é necessário fazer uma conversão, desta vez é preciso criar um novo *AppEvent,* registramos que em 100 rodadas de consensos gastamos 23ms [22].

Portanto, para calcular o valor aproximado gasto com a conversão dos eventos do iBusTFE para os do iBusTF no recebimento de uma mensagem remota, somamos o tempo gasto para converter uma mensagem a ser entregue para a classe *Newtop* e o tempo para converter uma mensagem relativa a um bloco completo a ser enviado para a *App_Consensus*. Assim, concluímos que para receber uma mensagem remota foram gastos, aproximadamente, 0,65ms com as conversões de mensagens visando permitir a utilização da classe Newtop.

- Outro fator importante, foi a alteração realizada na Thread *Delivery* que eliminou o bloqueio dela durante 50ms após a entrega de mensagens à *App_Consensus*. Esta alteração fez com que a Thread referida passasse a buscar ininterruptamente por blocos completos a

[22] O tempo registrado para criar os *AppEvents* com o uso das fábricas de objetos foi aproximadamente 5 vezes mais rápido do que para instanciar diretamente os *MessageEvent* do iBusTF. Isto ratifica a eficácia na utilização das fábricas de objetos (vide seção 4.2.1).

serem entregues para a aplicação. Como esta Thread necessita acessar algumas estruturas de dados compartilhadas (p.e.: o vetor LRV) com outras *Threads*, esperávamos que houvesse um impacto no desempenho, devido ao controle da concorrência, porém este impacto deveria ser proporcional utilizando as duas FCGs, pois ambas utilizam a Thread *Delivery*.

Após realizar os testes verificamos que o impacto gerado foi muito superior no iBusTFE do que no iBusTF. Percebemos que para executar o método *upHandleEvent,* ao receber uma mensagem remota, o iBusTFE demorou 692ms e o iBusTF 197ms durante as 100 rodadas de consensos. Ao investigar o problema averiguamos que a causa do alto tempo registrado no iBusTFE foi a ocorrência de 89 chamadas ao método `object.wait()` acarretando uma parada de 613ms da Thread *Delivery*. Com o iBusTF apenas houve 8 chamadas ao mesmo método, que implicaram em 21ms de espera.

Concluímos que a justificativa para esta diferença reside no fato que no iBusTF a Thread *Delivery* ao identificar algum bloco completo, executa toda a lógica para entregar esta mensagem à aplicação. Ou seja, esta Thread que executa o código da camada Stack e o da própria aplicação de consensos quando uma mensagem remota é recebida. No iBusTFE, a entidade *TotalOrderer* entrega as mensagens para a entidade *AppiBusTFE,* que possui um Thread própria, para que ela seja encaminhada para a aplicação [23].

Assim, no iBusTFE a Thread *Delivery* executa o código até que a mensagem seja entregue para a entidade *AppiBusTFE*. Deste ponto em diante, a Thread da entidade *AppiBusTFE* assume o controle e a Thread *Delivery* volta ao loop ininterrupto em busca de novos blocos completos.

Desta forma, constatamos que o escopo de atuação da Thread *Delivery* é menor no iBusTFE do que no iBusTF. Este fato faz com

[23] No iBusTFE para cada entidade consumidora ativa (p.e.: o *AppiBusTFE*) existe uma Thread responsável por receber os eventos e executar o código associado.

que a Thread *Delivery* execute mais vezes o código descrito no método `Delivery.run()` no iBusTFE, gerando uma maior concorrência pelas estruturas de dados compartilhadas.

Verificamos, com o auxílio do *Quantify,* que em 100 rodadas de consensos a estrutura de dados chamada de LRV (last received block), que armazena as informações sobre qual o último bloco recebido, foi acessada no iBusTFE 649.415 vezes e no iBusTF 379.297. Adicionalmente, registramos que os dois testes onde foram obtidos o número de acessos ao LRV duraram 2m23s e 2m29s com o iBusTFE e iBusTF, respectivamente, eliminando a possibilidade de que um teste pudesse ter sido mais longo, acarretando mais ciclos de execução do método `Delivery.run()`.

Considerando que a diferença gasta com o método `object.wait()` no método `upHandleEvent()` nas duas FCGs foi de 592ms, então podemos afirmar que a concorrência gerada pela forma que o iBusTFE foi implementado, a cada rodada de consenso, foi de aproximadamente 5,92ms por rodada de consenso. Como com um grupo de três membros a cada rodada devem ser processadas 2 mensagens remotas, concluímos que aproximadamente em média 2,96ms foram gastos adicionalmente no iBusTFE em função do aumento da concorrência no recebimento de cada mensagem remota.

Ressaltamos, que o problema da concorrência citado no parágrafo anterior se agrava com o aumento do número de membros remotos. Este fato foi observado em todos os grupos de testes e pode ser observado na Tabela 5.4. Com um grupo de 3 membros podemos observar que o desempenho das duas FCGs foi praticamente o mesmo, porém com o grupo de 5 membros o iBusTFE teve um desempenho parecido no tempo mínimo, mas no tempo máximo foi registrada uma diferença de 7ms. Notem que o tempo mínimo é obtido através da conclusão do primeiro consenso por um dos membros. Como o problema de concorrência, tipicamente, não ocorre em todos os consensos e em todos os membros ao mesmo tempo, então o tempo mínimo não foi muito afetado, porém o tempo máximo

denota que algum dos membros demorou acima do esperado para concluir o consenso. Portanto, a análise efetuada com o Quantify permitiu enxergarmos a causa do desempenho negativo do iBusTFE com grupo maiores utilizando a métrica do tempo máximo.

Após identificar e analisar os detalhes de implementação que interferiram negativamente no desempenho da componente TF do iBusTFE, resolvemos fazer uma análise geral considerando os tempos apurados com o Quantify. Nosso intuito foi de estimar como seria o desempenho do iBusTFE, subtraindo o tempo calculado como interferência de implementação.

Utilizando esta abordagem, obtivemos um tempo para o iBusTFE tratar uma mensagem remota de 11,31ms e do iBusTF de 12,71ms. Portanto, na nossa análise o iBusTFE foi, aproximadamente, 1,4ms mais rápido do que o iBusTF para receber uma mensagem remota. Assim, estimamos que com um grupo de 3 membros o iBusTFE seria aproximadamente 2,8ms mais rápido do que o iBusTF.

Verificamos que o tratamento seqüencial das mensagens efetuados pelo iBusTF, impõe um atraso no tratamento das mensagens de aproximadamente 2,24ms por mensagens remotas. Este valor foi obtido através da soma dos tempos gastos nas camadas REACH e NAK. Observe que isto implica que a entrega da mensagem à aplicação foi postergada, já que não existe paralelismo no modelo em camadas.

Não computamos o tempo gasto na camada FRAG como uma restrição do modelo em camadas, pois necessariamente há de se fazer este tratamento antes de entregar a mensagem para a aplicação. No iBusTFE esta operação é realizada pelo framework EVA e o seu tempo está embutido no tempo da camada IPMCAST.

Identificamos outras situações onde o paralelismo do modelo em camadas poderia favorecer o desempenho do iBusTFE. Por exemplo, no envio e recebimento de mensagens de *HeartBeat* pela camada NAK (vide seção 3.4.6.1) o iBusTFE envolve duas entidades para enviar e duas para receber as mensagens remotas. No caso do iBusTF, 3 camadas são envolvidas no envio e no recebimento. Neste caso, a camada REACH manipula as mensagens apesar de não fazer nenhum tratamento útil, ela age apenas como uma repassadora de mensagens. Este fato, apesar de não estar envolvido diretamente com o envio de mensagens com valores sugeridos influenciam negativamente o resultado do iBusTF, pois gera uma carga adicional de

processamento. Isto gera a concorrência pelo processador que em determinados momentos pode bloquear as *Threads* que estão envolvidas no envio das mensagens para a aplicação de consensos.

Notem que o problema de concorrência pelos recursos do sistema operacional entre as *Threads* criadas internamente pela FCG também ocorre no iBusTFE. Porém, neste caso apenas são envolvidas as *Threads* que executam o trabalho estritamente necessário, o que significa que os recursos do sistema operacional ficarão menos tempos alocados para processar um evento do tipo *HeartBeat.*

Outro ponto favorável ao iBusTFE, é que a camada TF foi implementada por duas entidades distintas *TotalOrderer* e *SuspectManager*, que provêem a ordenação total e a visão de grupo atômica, respectivamente. Assim, os eventos enviados para a aplicação de consenso não necessitam esperar pelo tratamento das mensagens pelo *SuspectManager*, para serem enviados para a aplicação de consenso.

No iBusTF, mesmo que a camada TF fosse dividida em duas não haveria nenhum ganho adicional, pois os eventos seriam manipulados seqüencialmente, ou seja, as mensagens remotas a serem enviadas para a aplicação de consensos, obrigatoriamente, necessitariam aguardar o tratamento de suspeitas de falhas de membros.

Portanto, podemos afirmar que o desempenho registrado nos testes com as FCGs foi equivalente, havendo na maioria dos casos um pequena diferença a favor do iBusTF. Esta pequena vantagem foi revertida quando estimamos qual seria o desempenho do iBusTFE sem as interferências dos detalhes de implementação que estavam influenciando seu desempenho.

5.3.2 Contribuições

Com a construção do iBusTFE e a execução das baterias de testes, levantamos dados que são relevantes para a comparação dos dois modelos. Porém, não conseguimos determinar qual o melhor modelo para o caso estudado. Relacionamos abaixo os principais fatos que impossibilitaram chegarmos a esta conclusão:

- Utilizamos como unidade de medida do tempo para comparar o desempenho das duas FCGs o milisegundo. Esta é a unidade de

tempo que o método java.lang.System.currentTimeMillis() fornece. Como o desempenho registrado com as duas FCGs foi de, aproximadamente, 4ms por consenso, diferenças de até 25% no desempenho das duas FCGs podem ter ocorrido e não foram identificadas devido à precisão utilizada.

- Em função dos baixos tempos obtidos, qualquer influencia de fatores externos, por menor que fosse, poderia interferir significativamente nos resultados. Observamos este fato ao longo da execução dos nossos testes. Dependendo dos ajustes realizados os resultados poderiam indicar que uma ou outra FCG obteve melhor desempenho.
- Não conseguimos controlar todos os fatores identificados que estavam interferindo nos testes. Quando não conseguimos elimina-los, adotamos uma estratégia de contorna-los ou então estimar o seu impacto e projetar qual seria o resultados sem interferências. Além deste fato, não podemos garantir que não havia outros fatores interferindo nos resultados de forma discreta, o que dificultaria a sua identificação.
- Como construímos o iBusTFE com base no iBusTF, podemos afirmar que ambos provêem as mesmas funcionalidades, porém não conseguimos garantir que os dois possuíssem a mesma implementação, pois este fato é impossível devido às diferenças existentes entre os dois modelos em que são estruturados. Estas diferenças de implementação podem ter influência nos resultados obtidos.
- A decisão de utilizar o framework EVA como a base para a construção do iBusTFE, impôs diferenças de implementação importantes, como a impossibilidade de implementar no iBusTFE um componente equivalente à camada FRAG, pois o EVA implementa as funcionalidade referentes a esta camada.

Ressaltamos que os fatores acima citados, dificultaram a nossa análise, porém conseguimos obter conclusões importantes na análise dos resultados, conforme se segue:

- A média para tratamento de uma mensagem foi de, aproximadamente, 4ms com as duas FCGs. Podemos afirmar que a média registrada é a soma do tempo gasto para executar a lógica das FCGs, para prover as propriedades que elas garantem e do tempo imposto pela forma como as FCGs foram estruturadas internamente. Desta forma, podemos constatar que o tempo gasto pelos dois modelos para estruturar a comunicação entre os módulos (componentes ou camadas) das FCGs foi muito baixo (menor que 4ms) .
- Constatamos que o tratamento seqüencial das mensagens imposta pelo modelo em camadas impõe um atraso na entrega das mensagens. Porém, este atraso apresentou no caso estudado, valores muito baixos (menores que 1ms), que não chegaram a ser representativos na análise final.
- Identificamos que os fatores que mais influenciaram no desempenho das duas FCGs, foram aqueles que não se referiam ao modelo de estruturação interna. Ou seja, caso os modelos sejam utilizados de forma apropriada, caso a implementação efetuada seja eficiente e livre de erros, independente do modelo utilizado, teremos protocolos de comunicação de alto nível com um bom desempenho.

Desta forma, podemos afirmar que conseguimos atingir o objetivo central deste estudo apenas em parte. Obtivemos dados importantes para a comparação dos dois modelos de estruturação interna, porém não foi possível constatar qual o melhor modelo. O problema da precisão utilizada e a existência de fatores externos não eliminados totalmente, impossibilitaram que pudéssemos afirmar qual modelo obteve melhor desempenho nos testes realizados. Assim, não podemos garantir que seriam obtidos dados proporcionais aos registrados neste estudo, caso mudássemos algumas configurações nas duas FCGs e executássemos novamente os testes. Acreditamos que não haveria grandes variações, mas mesmo assim não foi possível determinar o modelo com melhor desempenho.

Uma outra contribuição importante deste estudo, foi a de demonstrar que a tarefa de comparar dois modelos de estruturação interna em um ambiente distribuído não é simples. Para que sejam obtidos dados consistentes, devem ser

controlados diversos fatores que podem interferir nos resultados. Nesta pesquisa foi necessário estudar e controlar fatores distintos como tráfego de rede, configuração dos computadores envolvidos, controle de concorrência pelos recursos do sistema operacional, controle da coleta de lixo, sincronia entre os membros envolvidos nos testes etc. Além disto, deve-se possuir conhecimento profundo das implementações utilizadas, visando identificar e corrigir erros e / ou implementações não otimizadas que podem deturpar totalmente os resultados obtidos.

6 Conclusões e Trabalhos Futuros

Adotamos uma abordagem experimental para ressaltar as vantagens e desvantagens dos modelos em camadas e do baseado em canais de eventos, para a construção de protocolos de comunicação de alto nível. Para isso, construímos o iBusTFE, que é uma FCG baseada em eventos, a partir de uma FCG existente chamada iBusTF [CM00], construída seguindo o modelo baseado em camadas. Ambas FCGs disponibilizam os mesmos serviços para as aplicações.

Após construirmos o iBusTFE, iniciamos os trabalhos para comparar as duas FCGs, o critério escolhido para avalia-las foi o desempenho. Procuramos determinar como as duas FCGs se comportaram através de uma série de testes. A partir dos resultados, procuramos identificar qual a influência foi exercida pela forma de estruturação interna das duas FCGs.

De modo geral, podemos dividir o trabalho de comparação das duas FCGs em três fases: (1) preparação do ambiente, (2) execução dos testes e (3) análise dos dados. Foi necessário fazermos três iterações completas nestas fases, como o intuito de isolar diversos fatores que poderiam interferir nos resultados, para obter os dados a serem utilizados.

Na preparação do ambiente, planejamos quais testes seriam realizados, analisamos quais os fatores em potencial que poderiam afetar os nossos resultados e fizemos alterações necessárias para conseguir eliminá-los ou contorná-los, evitando que interferissem nos nossos trabalhos. Nesta fase as principais atividades executadas foram:

- Definição dos testes a serem executados.
- Montagem da rede de comunicação exclusiva para a realização dos testes previstos.
- Análise de todos os programas que ficariam ativos no momento dos testes.
- Homogeneização das configurações de hardware e software nos computadores envolvidos nos testes.
- Assegurar que o ambiente computacional estava adequado para a realização dos testes, através da verificação de diversos indicadores,

como por exemplo, tráfego de pacotes broadcast, número de interrupções de processador por segundo, taxa de utilização da CPU etc.

- Alterações diversas nas FCGs para controlar fatores como a desincronia no envio de mensagens, impacto do coletor de lixo (GC) nos tempos do consensos etc.
- Criação de programas para coletar os dados necessários para a fase de análise dos dados.

Conseguimos eliminar ou controlar a maioria dos fatores externos identificados, de modo que os resultados não fossem afetados. Porém, nos deparamos com fatores que estavam diretamente relacionados com detalhes de implementação das próprias FCGs, que não puderam ser alterados. Nestes casos, procuramos mensurar qual o impacto nos resultados e fizemos estimativas sobre como as FCGs deveriam se comportar de forma ideal, ou seja, sem a presença dos fatores remanescentes.

Na fase de preparação do ambiente geramos dois produtos utilizados nas fases seguintes: (1) a especificação dos testes a serem executados e (2) o próprio ambiente para a execução dos testes. Nos referimos a ambiente como o conjunto de equipamentos e sistemas necessários para executarmos os testes previstos de forma controlada, ou seja, eliminando os fatores adversos identificados.

A fase de execução dos testes foi a mais simples, utilizamos a especificação de testes e o ambiente preparado na fase anterior para executar as atividades desta fase. Monitoramos os testes para assegurar que eles fossem realizados nas condições idéias. Nesta fase, produzimos diversos arquivos de log que serviram como o insumo da fase de análise dos dados.

A partir dos arquivos de log gerados, analisamos o desempenho das duas FCGs. Na fase da análise de dados utilizamos três métricas distintas com o objetivo de identificar e entender o comportamento das duas FCGs nos diferentes testes que realizamos. Esta fase foi eminentemente investigativa, nela conseguimos identificar diversos fatores externos que estavam interferindo na nossa análise e quais as alterações necessárias para elimina-los. Foi comum gerar na fase de análise requerimentos para que uma nova fase de preparação de ambiente fosse realizada com o intuito de eliminar fatores adversos.

A cada iteração aprofundamos nossa análise, utilizando novas métricas e observando fatores que passaram despercebidos nas primeiras iterações. Ao final da última iteração obtivemos os dados finais, utilizados na análise das duas FCGs estudadas.

Apesar de termos calculado os tempos para a obtenção dos consensos de três formas distintas (mínimo do grupo, máximo do grupo e tempos individuais) e de todas as formas serem úteis para analisarmos o comportamento das FCGs, elegemos como a métrica de referência para nossa análise final o tempo mínimo do grupo para obtenção do consenso. Tivemos de adotar esta estratégia, pois identificamos que as outras métricas utilizadas foram afetadas por fatores externos.

O tempo máximo do grupo foi afetado no iBusTFE por problemas de concorrência, que foram gerados em função da utilização de mais de uma Thread para entregar as mensagens à aplicação. Por sua vez, o tempo individual para obtenção dos consensos estava sendo afetado por alguns picos de consensos de alto valor que surgiram utilizando as duas FCGs.

Assim, com base no tempo mínimo do grupo para a obtenção dos consensos como referência, pudemos identificar que as duas FCGs comparadas neste estudo registraram um desempenho praticamente igual. O iBusTF foi de modo geral um pouco mais rápido. Porém, a diferença máxima registrada a favor do iBusTF foi de 2ms e em média de apenas 1ms. Estimamos que esta pequena diferença seria revertida a favor do iBusTFE, caso não existissem detalhes de implementação alheios ao modelo de estruturação interna, interferindo nos resultados.

Acreditamos que caso fossem realizadas novas baterias de testes, este resultado poderia ser revertido para o iBusTFE em função da falta de precisão no agendamento do envio das mensagens de consensos, que estimamos ser em média de 1ms.

É importante ressaltar que alguns detalhes na implementação do iBusTFE inseriram atrasos não relacionados a forma de estruturação interna da FCG. Estimamos que estes atrasos tenham afetado o desempenho do iBusTFE, aumentado o tempo de processamento por consenso em torno de 25% (vide seção 5.3.1). Como a duração de uma rodada de consenso, foi em média de 4ms com este ganho o iBusTFE poderia se tornar em torno de 1ms mais rápido.

Considerando que o iBusTFE em condições idéias seria 1ms mais rápido, concluímos que seria mantida a equivalência nos resultados das duas FCGs. Mesmo considerando o ganho estimado do iBusTFE, no tempo mínimo a diferença entre o desempenho das duas FCG seria praticamente igual. Desta forma, ratificamos a conclusão de que o desempenho das duas FCGs no caso estudado foi equivalente.

O fato de que nossos experimentos foram executados em um tempo pequeno (média de 4ms por consenso), ressaltou o bom desempenho obtido com as duas FCGs. Desta forma, podemos considerar que ambas FCGs e , conseqüentemente, os modelos em que foram baseados são boas opções para a construção de protocolos de comunicação de alto nível.

Conseguimos demonstrar que o paralelismo e a eliminação do tratamento obrigatório das mensagens no modelo baseado em canais de eventos não ofereceu vantagens significativas a ponto de oferecer um desempenho diferenciado em relação ao modelo baseado em camadas.

Neste estudo demonstramos que a comparação entre dois modelos de estruturação interna em um ambiente distribuído não é simples. Existem diversos fatores que podem interferir nos resultados e que devem ser controlados. Nesta pesquisa foi necessário estudar e controlar fatores distintos como tráfego de rede, configuração dos computadores envolvidos, controle de concorrência pelos recursos do sistema operacional, controle da coleta de lixo, sincronia entre os membros envolvidos nos testes, entre outros.

Constatamos que os fatores alheios aos testes como, por exemplo, erros e/ou implementações não otimizadas poderiam alterar totalmente o resultado dos testes. Deste modo, foi muito importante analisarmos e otimizarmos as duas implementações utilizadas.

Analisando os dados obtidos, concluímos que ambos modelos de estruturação interna possuem um desempenho equivalente, inserindo atrasos menores que 4ms em média. Identificamos que o iBusTF inseriu um atraso devido ao tratamento seqüencial e obrigatório por todas as camadas das mensagens, porém este atraso foi baixo (menor que 1ms) e não interferiu significativamente nos testes.

A discussão a respeito das vantagens e desvantagens da aplicação do modelo de canais de eventos em detrimento do uso do modelo baseado em camadas deve ser fundamentada em estudos que verifiquem o comportamento de aplicações desenvolvidas com estes modelos. Neste estudo verificamos que a forma de estruturação interna das duas FCGs não foi um ponto determinante no que diz respeito ao desempenho. Diversos fatores (p.e.: erros de programação, coleta de lixo, falta de sincronia entre os membros etc) tiveram uma influência muito maior no desempenho das FCGs do que a forma de estruturação das FCGs. Assim, concluímos que ambos modelos são indicados para a construção de protocolos de comunicação de alto nível.

6.1 Trabalhos futuros

Não fazia parte do escopo deste estudo analisar outros critérios que também possuem suma importância no desenvolvimento de sistemas, como por exemplo, a flexibilidade, manutenibilidade e escalabilidade. Assim, é importante que haja novos estudos enfatizando outros critérios dos modelos baseado em camadas e em canais de eventos. Estes estudos poderão fornecer detalhes importantes sobre os modelos de estruturação, permitindo que haja maiores subsídios para decidir qual o modelo de estruturação interno mais apropriado para cada aplicação em particular.

Outra linha de estudo interessante é a de ampliar a comparação entre o iBusTFE e o iBusTF, mensurando o envio e recebimento de todos os tipos de mensagens gerados / consumidos por cada uma destas FCGs. Notem que no capítulo 5 nos concentramos em um caso particular de tratamento de mensagens (recebimento de mensagens remotas). Porém, há diversas outras situações (p.e.: recuperação de mensagens, exclusão de membros, envio de mensagens através de uma rede WAN etc) que podem ser mensuradas e enriquecerão os resultados.

Sugerimos também que sejam efetuadas algumas alterações na implementação do EVA com o propósito de facilitar o trabalho de projetar e desenvolver novas aplicações. Sugerimos que seja criado uma espécie de filtro para os produtores de eventos, visando permitir a entrega seletiva de eventos. Isto resolveria o problema da duplicação de eventos, permitindo que apenas a

comunicação do tipo “Componente X Componente” (vide capítulo 2) fosse utilizada, tornando o projeto mais simples e natural.

Desta forma, as entidades produtoras poderiam ter filtros associados à produção de mensagens para uma determinada entidade, permitindo que antes de enviarem as mensagens elas avaliem o interesse das entidades consumidoras registradas. Assim, os eventos a serem descartados não necessitariam ser tratados pelos consumidores, pois seriam descartados na origem. Na implementação atual a mensagem é entregue ao consumidor que antes de consumi-la verifica aplicando um filtro se deve descartar a mensagem ou não. Com a utilização de um filtro no produtor este processo poderia ser antecipado, evitando que o consumidor fosse notificado e, conseqüentemente, minimizando o custo de envio da mensagem entre o produtor e o consumidor.

Referências Bibliográficas:

[BBa03] Barretto, M., Brasileiro, F. iBusTF: Detalhes de Projeto e Implementação. http://lsd.dsc.ufcg.edu.br/publications, 2003.

[BBb03] Barretto, M., Brasileiro, F. iBusTFE: Detalhes de Projeto e Implementação. http://lsd.dsc.ufcg.edu.br/publications, 2003.

[BGH+a00] Brasileiro, F., Greve, F., Hurfin, M., Le Narzul, J. P., Tronel, F. Eva: an Event-Based Framework for Developing Specialised Communication Protocols. Proceedings of the IEEE International Symposium on Network Computing and Applications, p. 108-119, 2002.

[BGH+b00] Brasileiro, F., Greve, F., Hurfin, M., Le Narzul, J. P., Tronel, F. ADAM: A Group Communication System to Support Fault Tolerance. Reutel-2000 Document R317-D. December 2000.

[CHTCB95] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. Technical Report TR95-1548, Cornell University, October 1995.

[CM00] L. Cirne e R. Macêdo. *Uma Abordagem para Tolerância a Falhas em Java através de Comunicação em Grupo.* SBRC2000,18 Simpósio Brasileiro de Redes de Computadores, Belo Horizonte, maio de 2000.

[COM98] Comer. D.E. Interligação em rede com TCP/IP, Campus, 1998.

[EFGK01] Eugster. P.Th., Felber P., Guerraoiu R., Kermarrec A.-M. The Many Faces of Publish/Subscribe. Technical Report – January 2001.

[EG01] Eugster. P.Th., e Guerraoiu R. Content-Based Publish/Subscribe with Structural Reflection. In Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'01), January 2001.

[EGS01] Eugster. P.Th., Guerraoiu R. e Sventek J. *Type-Based Publish/Subscribe.* Technical Report TR-DSC-2000-029, Swiss Federal Institute of Technology, Lausanne, June 2000.

[FLS97] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. In *Proc. Of the Sixteenth ACM Symp. On Principles of Distributed Computing*, pages 53-62, Santa Barbara, CA, August 1997.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addisson-Wesley.

[GHO+00] Guerraoui, R., Hurfin, M., Mostefaoui, A., Oliveira, R., Raynal, M. , Schiper, A., Consensus in Asynchronous Distributed Systems: A Concise Guided Tour, Springer LNCS: Advances in Distributed Systems, N. 1752, pp. 33-47, 2000".

[GM98] F. Greve e R. Macedo. *The Membership Service Performance Analysis.* Anais do XVI Simpósio Brasileiro de Redes de Computadores – SBRC98, pp 682-700, Rio de Janeiro, Maio 1998.

[HOR97] Horstmann, Cay S. Pratical object-oriented development in C++ and Java. Wiley Computer Publishing, 1997.

[HR96] Mark Hayden and Robbert vanRenesse Optimizing Layered Communication Protocols Cornell University Technical Report, TR96-1613, November 1996.

[KOE99] P. Koenig. Messages vs. objects for application integration. Distributed Computing, 2(3):44–45, Abril 1999.

[LAM78] L. Lamport, *Time, Clocks and the Ordering of Events in a Distributed System.* Com. ACM, Dezembro 1978, vol. 35, n.12.

[LSP82] L. Lamport, R. Shostak e M. Pease. *The Bizantine Generals Problem.* ACM Transactions on Programming Languages and Systems 4,3, Julho 1982. pp, 382-401.

[MAC95] R. Macêdo. *BCG – Base Confiável de Comunicação em Grupo.* Projeto individual de pesquisa. CNPq. Bolsa de Produtividade de Pesquisa II-C número 300013/87-6, Janeiro, 1995.

[MC00] Microsoft Corporation. Performance Optimizing Tools. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/comsrv2k/htm/cs_mmc_optimizing_vwcq.asp, 2000.

[MCa01] Microsoft Corporation. Overview of Performance Monitoring. http://www.microsoft.com/windows2000/techinfo/reskit/en-us/default.asp?url=/windows2000/techinfo/reskit/en-us/prork/preb_mon_nbcl.asp, 2001.

[MCb01] Microsoft Corporation. Administrative tools. http://www.microsoft.com/windows2000/techinfo/reskit/en-us/default.asp?url=/windows2000/techinfo/reskit/en-us/prork/prda_dcm_gkfl.asp, 2001.

[MEM99] A. Marcel, M. Erzrberger e S. Maffeis. *iBus – A Software Bus for the Java Plataform.* JAVA Report. Setembro, 1999.

[MES93] R. Macêdo, P. Ezhilchelvan e 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.

[MGH98] Hayden, M. G. The Ensemble System. http://cs-tr.cs.cornell.edu/Dienst/UI/1.0/Display/ncstrl.cornell/TR98-1662. 1998

[MMSA+96] Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A. Budhia, R. K. Lingley-Papadopoulos, C. A. *Totem: A fault-tolerant multicast group communication system*. Communications of the ACM, 39(4), April 1996.

[POW96] D. Powell. *Group Communication.* Communications of the ACM, 39 (4), April 1996, 50-53. Guest Editor, special issue on Group Communication.

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

[VKCD99] Vitenberg, R., Keidar, I., Chockler, G. V., Dolev D. Group Communication Specifications: A Comprehensive Study. http://citeseer.nj.nec.com/vitenberg99group.html. 1999.

[OPSS93] Oki, B., Pfluegl, M., Siegel, A., e Skeen, D. The information bus: an architecture for extensible distributed systems. In *Proc. of the 14 ACM Symposium on Operating Systems Principles* (Asheville, USA, Dec 1993), pp.58-68.

[RAT02] Rational Software Corporation. *Getting Started with Rational Quantify.* http://www.rational.com/support/documentation/manuals/.

[SCH90] F.B. Schneider. *Replication management Using the State Machine Approach.*ACM Computing Surveys, Pg22. Dezembro 1990.

[SKE98] Skeen. D. Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview. http:/www.vitria.com, 1998.

[SCH90] F.B. Schneider. *Replication management Using the State Machine Approach.*ACM Computing Surveys, Pg22. Dezembro 1990.

[SUN02] Sun Microsystems, Inc. JavaTM 2 Platform, Standard Edition, v 1.3.1

API Specification. http://java.sun.com/j2se/1.3/docs/api/index.html.

[SUN99] Sun Microsystems Inc. *Tuning Garbage Collection with the 1.3.1 JavaTM Virtual Machine.* http://java.sun.com/docs/hotspot/gc/, 1999.

[TAN95] Tanenbaum, A.S. Sistemas Operacionais Modernos. Guanabara Koogan, 1995

[TAN96] Tanenbaum, A.S. Computer Networks – Third Edition. Prentice Hall, 1996