UNIVERSIDADE ESTADUAL DE CAMPINAS

INSTITUTO DE COMPUTAÇÃO

Proposta de Projeto – Disciplina de MO441

DESENVOLVIMENTO DE UMA PLATAFORMA PARA SIMULAÇÃO DE ALGORITMOS DE BROADCAST

Anderson Rocha

Carlos Scussiatto

Leyza Baldo Dorini

Marcelo Rigon

Resumo

O desenvolvimento de protocolos de *broadcast* confiáveis está intimamente relacionado com sistemas distribuídos e de tolerância a falhas. A literatura nos apresenta várias implementações possíveis nas mais variadas combinações de custos e garantias de confiabilidade. Neste contexto, a presente proposta mostra-se de fundamental importância. Uma plataforma para simulação de algoritmos de *broadcast* apresenta-se como uma solução não só didática, permitindo aos interessados aprender sobre o funcionamento de uma abordagem em particular, como também como uma plataforma para constante melhoria de algoritmos em desenvolvimento. A plataforma servirá também como base de simulação de novos algoritmos antes destes serem implementados para resolver algum problema em particular.

1 Introdução

Broadcast é um dos pontos mais fundamentais de um sistema distribuído e de tolerância a falhas. É o bloco fundamental em que muitas outras abordagens se apoiam para resolver seus problemas. Em geral, um algoritmo de *broadcast* é um algoritmo para disseminação de uma informação (mensagens) por uma dada rede. Esta definição é muito ampla e pode incluir sistemas que podem ou não garantir o recebimento destas mensagens. Quando falamos em *broadcast confiável* estamos garantindo que, mesmo na presença de falhas, os devidos receptores da mensagem a receberão. Quando falamos em *broadcast confiável e atômico* acrescentamos a garantia de que o recebimento se dará dentro de um certo limite de tempo conhecido [7].

Há muitas formas de se implementar algoritmos de *broadcast*. Desde a maneira mais simples possível em que um dado iniciador faz a sua disseminação por *flooding* até os sofisticados sistemas baseado em *acknowledgement* implícitos que reduzem grandemente a banda de comunicação de mensagens entre os participantes.

De forma geral, ao se implementar um protocolo de *broadcast* deve-se levar em conta um acordo entre o número total de mensagens, a latência do sistema e a performance do protocolo em cenários sem falhas bem como em cenários com falhas. É importante deixar claro que não existe uma implementação de *broadcast* universal. Não existe um algoritmo mágico que funciona e resolve todos os casos. É neste ponto que este trabalho torna-se de fundamental importância. O desenvolvimento de um sistema para simulação de algoritmos de *broadcast* irá permitir que várias abordagens sejam comparadas e que a que mais se adeque a um determinado

problema possa ser utilizada. Objetiva-se que esta plataforma permita caracterizar vários cenários difentes sob as mais diversas circunstâncias possíveis. O usuário poderá conhecer rapidamente o comportamento de um dado protocolo de *broadcast* no quesito tempo de término, confiabilidade, número total de mensagens, robustez a falhas, recuperação após *crashes* entre outras facilidades [3] e [1].

2 Terminologia

2.1 Broadcasts

Não há uma especificação simples para *broadcast* que sirva para todos os casos. Diferentes especificações, mais ou menos restritivas, podem ser feitas. O acordo (*agreement*) que deve existir entre os processos é uma propriedade que captura o conjunto de mensagens que todos os processos não defeituosos de um sistema precisam entregar. Desta forma, podemos definir duas abordagens de *acordo* [7]:

- **Acordo não uniforme**: requer que todos os processos não defeituosos entreguem o mesmo conjunto de mensagens.
- **Acordo uniforme**: requer que todos os processos não defeituosos também entreguem mensagens que lhes foram entregues por outros processos antes daqueles se tornarem defeituosos.

Um protocolo de *broadcast* pode ser assíncrono ou síncrono. Um protocolo é dito síncrono quando ele possui um tempo de funcionamento previamente conhecido tanto para o seu término como para cada uma de suas fases intermediárias. Isto implica que o tempo de recebimento de mensagens é conhecido. Já um protocolo é dito assíncrono quando estes tempos não são conhecidos ou limitados. As mensagens são ditas que serão eventualmente entregues em algum momento no futuro. Cada uma destas possibilidades ainda pode ser estendida para levar em conta o conceito de atomicidade. Um protocolo é dito *atômico* e *síncrono* quando ele garante a existência de uma constante de tempo Δ tal que, mesmo na presença de até f falhas no processo de *broadcast*, as seguintes propriedades são satisfeitas:

- *Atomicidade*: se qualquer processador entrega uma mensagem no tempo U em seu $clock$ então todas as mensagens de *broadcast* iniciadas nesse tempo também serão entregues por todos os processadores corretos.
- *Ordem*: todas as mensagens entregues pelos processadores são entregues na mesma ordem.
- *Terminação*: todas as mensagens que tiveram seu *broadcast* iniciado no tempo T serão entregues no tempo $T + \Delta$;

As propriedades de *atomicidade* e *ordem* garantem que as mesmas mensagens serão entregues para todos os processadores corretos no sistema na mesma ordem. Desta forma, se o sistema encontra-se inicialmente consistente ele permanecerá consistente. O tempo de terminação Δ garante que todos os processadores corretos apresentam os mesmos resultados após cada passo de Δ unidades de tempo em seus $clocks$. Por outro lado, um protocolo é dito atômico assíncrono quando ele sacrifica o tempo de terminação com o objetivo de permitir maior realidade nas circunstâncias do meio e para tolerar falhas de performance nos *links* de comunicação entre

os processadores envolvidos. Desta forma, não há uma garantia no tempo de terminação ou mesmo no tempo de troca de mensagens entre os processadores [4].

Os protocolos de *broadcast* atômico são muito importantes para aplicações críticas de tempo real que precisam garantir limites nos tempos de resposta mesmo quando alguns componentes possam falhar. Estes limites são conseguidos assumindo que os atrasos de mensagens entre os processadores corretos no sistema são limitados e que há caminhos de comunicação redundante o suficiente entre os processadores. Esta suposição de que os atrasos entre os tempos de transmissão de mensagens são conhecidos implica que os sistemas de *broadcast* atômicos síncronos sejam implementados por *Sistemas Operacionais de Tempo Real*. Por outro lado, os sistemas de *broadcast* atômicos assíncronos, por não possuírem esta necessidade do conhecimento prévio dos tempos de atraso entre mensagens, podem ser implementados por sistemas operacionais de tempo compartilhado. No entanto, estes protocolos são indicados apenas para aplicacações em que o custo de perder algumas mensagens não é tão significativo [7] e [1].

A plataforma de simulação de algoritmos de *broadcast* será genérica e, desta forma, tem que admitir a combinação de várias políticas de garantia. Por exemplo, ao tratarmos de *broadcast confiável*, temos que garantir a satisfação das três propriedades básicas de um *broadcast confiável*: integridade, validade e acordo definidos sobre o broadcast em si e sobre os eventos de entrega das mensagens. Supondo que haja uma separação entre o evento de fazer um *broadcast* e o evento de entregar uma mensagem temos:

1. **Integridade**: se um processo correto fizer um *broadcast* de uma mensagem m, então ele eventualmente entregará m.

2. **Validade**: para qualquer mensagem m, todo processo entrega m no máximo uma vez, e apenas se m teve um evento de *broadcast* iniciado por algum processo.

3. **Acordo**: pode ser uniforme ou não uniforme como descrito anteriormente.

   - *uniforme*: se um processo entrega m, então todos os processos corretos entregam m;
   - *não-uniforme*: se um processo correto entrega m, então todos os processos corretos entregam m,

As propriedades acima poderão ser configuradas de acordo com as definições de cada protocolo. Por exemplo, caso o protocolo sendo testado seja o de *broadcast* por inundação (*flooding*) de mensagens, pode ser interessante alterar a propriedade de validade para que o processo reencaminhe todas as mensagens que ele receber e não apenas uma delas.

2.2 Cenários de falhas

Os protocolos que implementam *broadcast* de forma geral precisam lidar com os mais variados cenários de falhas possíveis. Uma classificação aninhada dos cenários de falhas [5] é dada a seguir e será utilizada na plataforma na simulação do Algorimo 1 (Seção 5). Uma falha por *omissão* ocorre quando um componente omite a resposta a um evento de entrada; uma falha de *crash* ocorre quando após uma primeira omissão, um componente sistematicamente omite respostas aos eventos de entrada subseqüentes até seu reinício. Uma falha de *temporização* (*timing*), ocorre quando um componente ou omite uma resposta ou responde a um determinado evento muito cedo ou muito tarde. A maioria das falhas de temporização observadas no mundo real envolvem

falhas por respostas atrasadas. A Figura 1 apresenta a relação entre estas falhas. Falhas de *crash* são uma subclasse de falhas falhas por *omissão* que, por sua vez, são uma subclasse de falhas de *temporização* e estas são uma subclasse das falhas *arbitrárias* ou *bizantinas*.

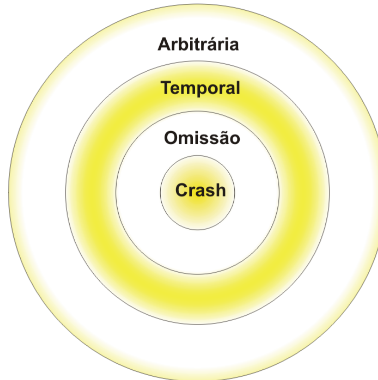

Figura 1: Cenário aninhado de falhas

Para a simulação do Algoritmo 2 (*pbcast*, Seção 6), o cenário de falhas será distinguido em duas categorias: *Hard failures* e *Soft failures* como apresentado por [2]. As falhas do tipo *hard* são *crashes* ou particionamentos da rede. As falhas *soft* são perdas das mensagem que foram corretamente recebidas (devido a *buffer overflow*, por exemplo), erros nos limites para receber mensagens de entrada ou instabilidades na rede que violam localmente as propriedades de *throughput* e *reliability*. Apesar do protocolo *pbcast* não tolerar falhas bizantinas, ele tolera falhas do tipo *hard* e também uma quantidade limitada de falhas do tipo *soft*. A idéia de tolerar esse último tipo de falhas está em superá-las com um mínino de impacto no *throughput* dos *multicasts* enviados de processos corretos para outros processos corretos.

3 Objetivos e metodologia

3.1 Objetivos

Este projeto visa atender às seguintes tarefas:

- estudar os principais algoritmos de *broadcast* disponíveis na literatura;
- identificar as vantagens e desvantagens de tais técnicas;
- buscar, na literatura, e/ou propor possíveis soluções para minimizar estas desvantagens;
- desenvolver uma ferramenta (*software*) onde será possível acompanhar o processo de algumas técnicas de *broadcast*. Pretende-se implementar algumas destas técnicas além de propor formas híbridas de modo a diminuir as fraquezas conhecidas em outras técnicas do tipo. Deste modo, objetiva-se criar uma ferramenta robusta que possa servir efetivamente aos pesquisadores e aos estudantes como um meio eficaz no estudo de algoritmos de *broadcast*;

- disponibilizar todo o material bibliográfico utilizado para o desenvolvimento da pesquisa. Desta forma, há a divulgação dos estudos do andamento da pesquisa. Com esse objetivo, será construída uma página (site) na rede mundial de computadores (*internet*);

3.2 Metodologia

Este projeto atenderá aos seus objetivos utilizando-se dos seguintes métodos:

- **Levantamento bibliográfico**. Será feito um levantamento de artigos científicos clássicos e atuais relacionados à *broadcasts* e *sistemas distribuídos*.
- **Estudo das principais técnicas de *broadcast*** através dos artigos coletados.
- **Implementação**. Dá-se início à implementação da plataforma de simulação de algoritmos de *broadcast*. A preocupação de construir códigos-fonte manuteníveis será constante. O paradigma de programação a ser utilizado é a orientação a objetos e a linguagem de programação ainda não foi definida.
- **Validação**. Terminada a implementação da plataforma, passa-se para a etapa de testes em laboratório com o uso inicialmente das duas técnicas apresentadas neste projeto.
- **Documentação**. Será desenvolvida uma documentação das técnicas desenvolvidas para posterior divulgação em artigos científicos e na *internet*.
- **Artigos, dissertação e relatórios**. Ao término do projeto, artigos serão elaborados para divulgação através da submissão à eventos e periódicos científicos relacionados ao tema. Além disso, um produto de *software* e uma dissertação de mestrado são esperados como resultado final do projeto.

4 Descrição da plataforma

A plataforma a ser implementada deverá possuir as seguintes características:

- Permitir a configuração do número de processadores participantes no cenário. Cada processador poderá ser definido independentemente tendo um número de processos próprio.
- Serão disponibilizadas primitivas básicas de comunicação como SEND e RECEIVE disponíveis para um processo interno a um processador e *send* e *receive* disponíveis aos processadores.
- Cada processador pode possuir *buffers* de envio e recebimento de mensagens que por sua vez podem ou não possuir falhas.
- Cada *buffer* terá implementado lógica controle de canais, códigos corretores de erro para descartar mensagens com erro de canal entre outras necessidades ainda não levantadas.
- Os canais de comunicação são independentes, ou seja, podem ter características próprias como tempo de atraso e probabilidade de omitirem ou não uma mensagem.

- Cada processador pode ter acesso a uma primitiva de agendamento para que ele agende uma tarefa a ser realizada em algum momento específico no futuro.
- O número de canais de comunicação entre os processadores participantes e o número máximo de falhas suportado pelo protocolo sendo testado deve ser configurável. Cada operação do sistema tem seu tempo configurado separadamente.
- O sistema deve funcionar de forma transparente ao usuário podendo simular tanto um sistema operacional de tempo compartilhado para algoritmos de *broadcast* assíncronos como simular um sistema operacional de tempo real respeitando cada agendamento feito por um evento do sistema.
- Um processador poderá ou não voltar de um *crash*, sendo possível também configurar o número máximo de vezes em que isso é permitido. Pode-se ter acesso a relatórios completos do sistema de modo a saber as mensagens devidamente entregues, perdidas ou ainda em andamento. O tempo de cada operação também pode ser verificado.
- Será possível configurar uma memória de eventos em cada processador de modo que estes possam armazenar mensagens recebidas e possam efetuar operações sobre estas antes de entregá-las. Os tipos de operações possíveis sobre um determinado conjunto de mensagens também poderá ser configurado.
- Cada processador poderá ou não manter um *log* de todos os eventos que aconteceram com ele até um determinado tempo.
- Poder-se-á definir um processador central capaz de controlar todas as ações do sistema ou caracterizar o sistema como totalmente distribuído.
- Será possível a sincronização de relógios para algoritmos que precisarem deste requisito inicial.
- Será possível configurar a que distribuição estatística o número e o tipo de falhas possíveis deverão obedecer: *distribuição gaussiana* segundo algum desvio padrão e média, *poisson*, *t-student* entre outras.
- Toda a configuração deverá se dar por meio de uma interface gráfica amigável e/ou através de arquivos de configuração para quando o número de processos for muito grande para ser caracterizado visualmente.

Muitas outras características poderão ser adicionadas à medida que se tornem conhecidas e necessárias. A validação inicial da plataforma será feita com a implementação e comparação teórico-prática dos Algoritmos 1 e 2 apresentados a seguir nas Seções 5 e 6.

5 Algoritmo 1 - Synchronous Atomic Broadcast for Redundant Broadcast Channels

Neste capítulo apresentamos um protocolo de broadcast atômico síncrono para sistemas distribuídos de tempo-real. O protolo é baseado em canais de *broadcast* redundantes e pode tolerar até f falhas de componentes, as quais serão detalhadas no decorrer do texto.

5.1 Modelo do sistema

Consideramos um sistema onde os processos mantém cópias replicadas do estado da informações, sendo que atualizações destas são disseminadas utilizando um serviço de *broadcast* atômico implementado para n processadores distribuídos. Tais processadores possuem nomes distintos e ordenados. Estão ligados a $f+1$ canais independentes através de $f+1$ adaptadores, também independentes. A Figura 2 ilustra o modelo descrito.

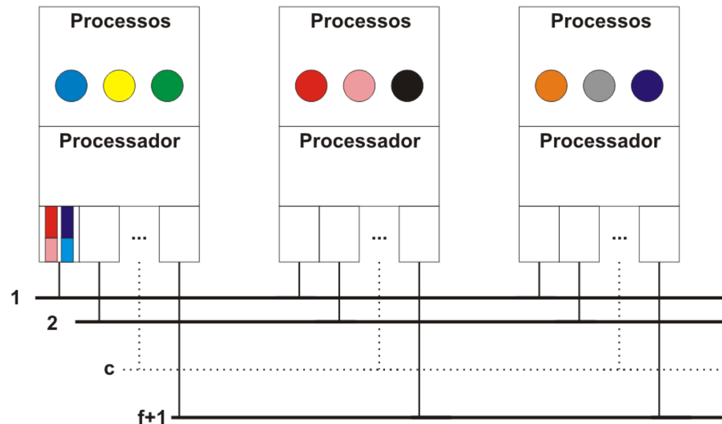

Figura 2: Modelo do sistema

Quando um processo deseja transmitir uma mensagem σ ele invoca um comando *SEND*, através do qual ele avisa ao processador que deseja fazer um *broadcast*. Da mesma forma, quando deseja receber um *broadcast*, ele se comunica com o processador através do comando *RECEIVE*.

Cada um dos $f+1$ adaptadores possui um *buffer* de memória (que é onde a mensagem aguarda até que seja colocada no canal), e lógica tanto para controlar o canal ao qual está ligado quanto para enviar e receber mensagens deste canal. É conveniente entender um adaptador como sendo composto por duas metades: um *in-adapter* que auxilia no processo de comunicação no sentido canal $\rightarrow$ processador, e um *out-adapter* que recebe mensagens do processador e as repassa ao canal correspondente para transmissão. Um processador se comunica com os adaptadores através dos comandos *send* e *receive*.

Podemos resumir o funcionamento do modelo da seguinte forma: se um determinado processo *p1* decide transmitir uma mensagem σ, ele invoca um comando *SEND* ao processador relacionado p, que por sua vez invoca um comando *send* a cada um dos $f+1$ *out-adapters*. Quando cada *out-adapter* consegue controle sobre o canal ao qual está relacionado, ele insere σ neste. Então, cada canal entrega a mensagem a todos os *in-adapters* a ele relacionados (um em cada processador). Para receber σ dos *in-adapters*, cada processador invoca um comando *receive*. Os processos, por sua vez, disparam comandos *RECEIVE* para receber σ do processador correspondente.

5.2 Suposições

São oito as suposições:

1. a taxa de *broadcast* é limitada: a velocidade à qual as mensagens são geradas é limitada pela capacidade que tanto os processos quanto os processadores têm de recebê-las;

2. os canais sofrem apenas falhas de omissão: são duas as formas que um canal c pode se comportar. Na ausência de falhas, todos os *in-adapters* ligados à c recebem uma determinada mensagem m dentro de um certo tempo constante C. Quando há falhas em c, pode ser que apenas um subconjunto (podendo inclusive ser vazio) de *in-adapters* receba m. Isso porque as mensagens corrompidas são detectadas pelos próprios protocolos de transmissão, podendo assim ser descartadas pelos *in-adapters*;

3. os *out-adapters* sofrem apenas falhas de desempenho: seja a constante O o tempo decorrido entre o momento que um processador coloca uma mensagem m em um *out-adapter* o e o momento em que o inicia a transmissão de m. Devido a uma falha de desempenho, pode ser que o tempo seja maior que O. Devido a essa possível falha, um *out-adapter* o pode se comportar de duas maneiras em relação à resposta de um comando *send(m)*. No primeiro caso, o consegue sucesso, colocando m em c dentro do tempo O. Caso haja uma falha de desempenho, pode ser que m seja enviada mais tarde, ou então que nunca seja enviada;

4. os *in-adapters* sofrem apenas falhas de omissão: o atraso que ocorre entre o momento que um canal entrega uma mensagem m para um *in-adapter* e o momento que m é recebida pelo processador correspondente é limitado em tempo por uma constante I. Assim como nos casos anteriores, um *in-adapter* pode se comportar de duas maneiras: ou entrega m dentro do tempo I ou nunca entrega m;

5. os processadores podem sofrer somente falhas de *crash*: P representa tanto o atraso entre o momento que um processador p recebe a solicitação de envio de uma mensagem m por um processo até o momento em que p coloca m em todos os $f + 1$ *out-adapters*, quanto o tempo decorrido entre o momento que um adaptador está pronto para entregar m para p e o momento que p recebe e processa m. Sobre tais suposições, um processador p pode proceder de duas formas ao interpretar comandos *SEND*(σ). A primeira delas é que, dentro de um tempo P, p consegue enfileirar as mensagens contendo σ em todos os seus *out-adapters*. No entanto, pode ser que p sofra uma falha do tipo *crash* e consiga enfileirar as mensagens em apenas um subconjunto (possivelmente vazio) de mensagens. Neste último caso, m nunca é enfileirada nos demais *out-adapters*;

6. processadores têm acesso à *clocks* corretos que estão aproximadamente sincronizados: assumimos que os processadores estão aproximadamente sincronizados, diferindo no máximo por uma constante ϵ;

7. tarefas podem ser programadas para certos *deadlines*: através do comando *Agende A(B) em T* é possível agendar a execução de uma tarefa A, de parâmetros B, no tempo T. Vale ressaltar que múltiplas invocações de um mesmo comando *Agende* tem o mesmo efeito de uma única invocação, e que uma chamada à *Agende A(B) em T* em um tempo $U > T$ é ignorada;

8. no máximo $f \leq n - 2$ componentes podem falhar durante um *broadcast*: consideramos como componentes processadores, adaptadores ou canais. A suposição 8 garante que apesar de haver até f falhas de componentes, pelo menos um caminho continuará válido entre dois processadores corretos p e q.

5.3 Idéia básica: lazy forwarding

As suposições de 2 a 6, apresentadas na seção anterior garantem que, na ausência de falhas, qualquer mensagem σ aceita para *broadcast* por um determinado processador é recebida e processada pelos demais processadores

dentro de um tempo $T + \delta + \epsilon$, onde T é o *timestamp*, δ é o atraso da comunicação entre processadores $(P + O + C + I + P)$ e ϵ é a diferença máxima entre os clocks dos processadores.

Entretanto, na presença de falhas pode ser que um subconjunto de processadores (com número de elementos ≥ 1) não receba uma mensagem m dentro do tempo $T + \delta + \epsilon$, e os demais processadores recebam. Isto representa um problema, pois viola a propriedade da atomicidade, apresentada anteriormente. Um cenário em que isso ocorre é dado na Figura 3.

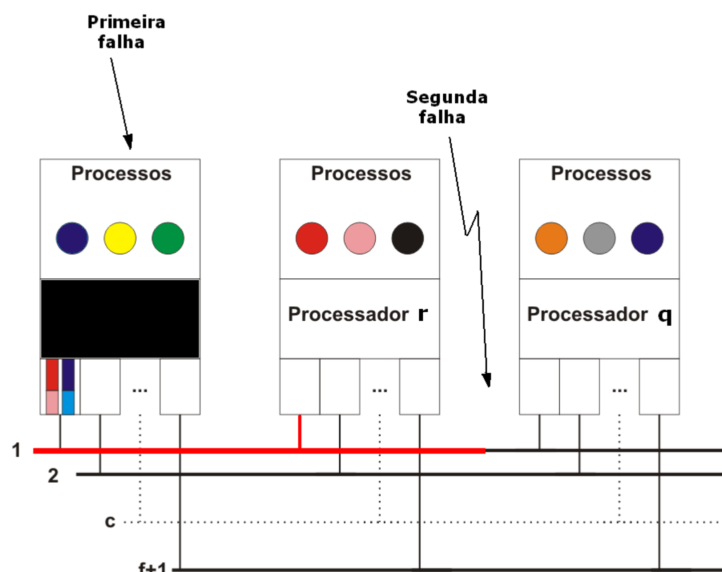

Figura 3: Pior cenário para duas falhas

O que acontece no cenário acima é que o processo que está iniciando o *broadcast* falha após colocar a mensagem m apenas no *out-adapter* do primeiro canal. A segunda falha corrompe m depois que ela foi entregue ao processador r e antes de chegar a q.

Para contornar esta situação, podemos aproveitar o fato de que o processador r recebeu m, fazendo que que ele encaminhe (*forward*) novamente m. Uma possível regra para este encaminhamento é dada pela *prompt forwarding rule*, que determina que um processador encaminhe a mensagem que recebeu de um dado canal em todos os f canais restantes. O número total de mensagens enviadas é $(f + 1) + (n - 1)f$ (destas, $(f + 1)$ são enviadas pelo processador iniciador e f são enviadas por cada um dos $n - 1$ processadores restantes).

Entretanto, essa abordagem realiza muito trabalho desnecessário, enviando um grande número de mensagens mesmo quando não ocorrem falhas. A idéia da *lazy forwarding rule* é justamente fazer com que sejam feitos encaminhamentos apenas quando necessário. Aqui apresentaremos uma versão inicial da regra, que será refinada no decorrer do texto.

> *Simple lazy forwarding rule*: um processo iniciador s enfileira qualquer mensagem m a ser transmitida nos *out-adapters* para os canais $1, 2, \ldots, f + 1$ em ordem crescente. Seja p um processador arbitrário, diferente de s que recebe m e seja c o maior número de canal no qual p recebe uma cópia de m. Se $c \geq f$, então p não precisa encaminhar m. Caso contrário, ou seja, $c < f$, p deve encaminhar m nos canais $c + 1, \ldots, f$.

A seguir, ao provarmos a propriedade da *unanimidade*, veremos que a regra é correta. A propriedade da unaminidade garante que se um processador s inicia a transmissão de uma mensagem m e algum processador correto p recebe m, então qualquer processador correto q recebe m. A prova é dada na seqüência.

Vamos considerar dois casos: $c \geq f$ e $c < f$. No primeiro caso, quando $c > f$, pode-se concluir que s conseguiu enfileirar a mensagem m em todos os $f+1$ *out-adapters* (considerando que as mensagens são enfileiradas em ordem crescente). Com isso, mesmo que f falhas ocorram, todos os processadores corretos certamente receberão m.

Podem ser dois os casos em que $c = f$. No primeiro, s conseguiu colocar m nos $f+1$ *out-adapters* e ocorreu uma falha que impediu o processador r de receber a mensagem pelo canal $f+1$. No entanto, já concluimos no parágrafo anterior que quando s consegue enfileirar m nos $f+1$ *out-adapters* a propriedade da unaminidade é garantida. O outro caso em que $c = f$ é quando s enfileira m nos f primeiros canais e então falha. Aqui a chave é perceber que devido à falha de s existem apenas mais $f-1$ falhas possíveis. No entanto, essas $f-1$ falhas não impedem que q receba m. Perceba que até este momento ($c \geq f$) não foi necessário fazer adiantamento.

O último caso, quando $c < f$, é possível apenas quando ocorre pelo menos uma falha antes do processador correto r receber m: pode ser que s tenha falhado ou então que um canal de número maior que c (ou o adaptador relacionado) tenha tido problemas. Quando isso ocorre, pode ser que um processador correto venha a não receber m. No entanto, como r encaminha m para os canais $c+1, \ldots, f$, as falhas restantes (que são de número menor que $f-1$) não impedirão os processadores corretos de receber m em pelo menos um de seus canais.

Dessa prova, podemos tirar algumas informações importantes. Primeiro que a *lazy forwarding rule* realmente só encaminha mensagens quando é necessário. Outra constatação é que ela envia apenas $f+1$ mensagens (na ausência de falhas), que é o número mínimo de mensagens necessárias quando *atomicidade* e *terminação* precisam ser garantidas. A prova pode ser encontrada em [4].

5.4 Descrição detalhada do protocolo

Para apresentar o protocolo proposto, primeiro abordaremos o caso em que uma única falha ocorre. Embora não haja necessidade de realizar encaminhamento neste caso, ele é importante para facilitar o entendimento do protocolo para múltiplas falhas.

5.4.1 O protocolo tolerante à uma única falha

Neste protocolo, cada mensagem possui o formato (T, s, σ), ou seja, carrega seu *timestamp*, o nome de seu iniciador e a atualização. Para garantir a propriedade de ordem exposta anteriormente, temos que:

- cada processador entrega as mensagens que recebe segundo o *timestamp* destas. Caso existam *timestamps* iguais, o desempate se dá pelo nome do processador iniciador. Vale observar que como os nomes dos processadores são distintos, a dupla (T, s) identifica unicamente um *broadcast*;

- deve-se garantir que um processador só entrega uma mensagem com *timestamp* T após o tempo $T+\delta+\epsilon$, que é o tempo limite para que uma mensagem com *timestamp* menor ou igual a T chegue. A partir

daqui, chamaremos $\Delta = \delta + \epsilon$ de *tempo de terminação* do protocolo, e $T + \Delta$ de tempo de entrega de atualizações cujo *timestamp* é T.

As mensagens recebidas por um processador p são processadas e armazenadas em um histórico (H) local a p, até que p as entregue aos seus processos. Para manter este histórico finito, eliminamos as atualizações de H assim que elas são entregues aos processos. Além disso, é realizado um "teste de aceitação de mensagem atrasada" (*late message acceptance test*), o qual verifica se a mensagem recebida não possui um tempo maior que o tempo de entrega. Assim, se U é o tempo local, uma mensagem é descartada caso $U \geq T + \Delta$.

Vamos agora discutir o protocolo de forma detalhada. Cada processador executa três tarefas diferentes, conforme ilustrado na Figura 4.

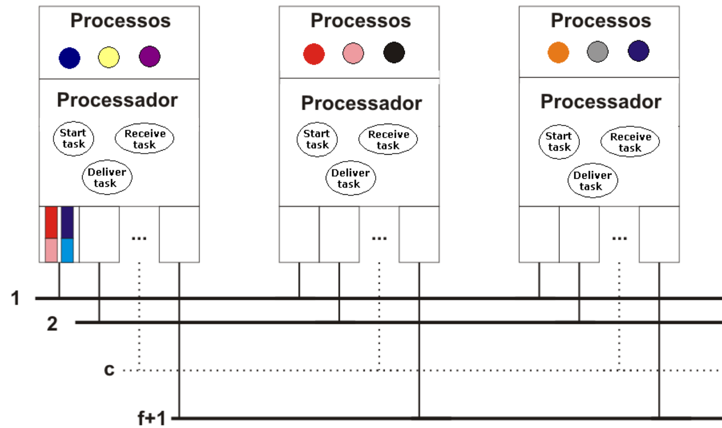

Figura 4: Tarefas concorrentes executadas por cada processador

A tarefa de início (*start task*) é quem vai iniciar o *broadcast* atômico (faz a comunicação processo $\rightarrow$ processador e processador $\rightarrow$ canal), a tarefa de recebimento (*receive task*) é que vai receber as atualizações (comunicação canal $\rightarrow$ processador) e, por fim, a tarefa de entrega (*deliver task*) vai de fato "entregar" as atualizações aos processos que invocam comandos *RECEIVE* (comunicação processador $\rightarrow$ processo).

Após cada algoritmo de tarefa comentaremos o código, visando explicar o que está acontecendo. Iremos nos referir ao algoritmo i linha j como (i, j).

Conforme exposto, a tarefa de início será ativada quando um processo invocar um comando *SEND*, que irá ativar a entrada $\overline{\textit{SEND}}$ (1.4). O algoritmo grava em T o momento em que recebeu (e aceitou) a atualização σ para broadcast (1.4). Invocando comandos *send*(T, *meuID*, σ), o processador solicita a transmissão da mensagem aos *out-adapters* dos canais 1 e 2 (1.5). Armazena-se no histórico H através do operador $\oplus$ (atualização) o fato que o processador identificado por *meuID* fez um *broadcast* da mensagem σ no tempo T (1.7). Por fim, agenda-se a entrega de mensagens com *timestamp* T no tempo $T + \Delta$ (1.8).

Na tarefa de recebimento, os processadores irão se comunicar com os *in-adapters* através do comando *receive*, socilitando o recebimento de mensagens dos canais (2.4). Para que seja possível realizar o *late message acceptance test*, o $clock$ é gravado na variável U (2.5).

Conforme exposto anteriormente, neste teste será verificado se o tempo local (U) não é maior que o tempo de entrega $(T + \Delta)$ (2.5), ou seja, se o tempo limite para que a mensagem chegasse já não terminou. Outro

Algoritmo 1 Procedimento para Iniciar

1: **função** TASK START
2: **const** $\Delta = \delta + \epsilon$;
3: **var** T: Tempo; σ: Mensagem; s: Processador;
4: **enquanto** *ciclo* **faça** $\overline{SEND}(\sigma)$; T ←*clock*;
5: **para** $c = 1, 2$ **faça**
6: *send*(T, *meuID*, σ) sobre c;
7: $H \leftarrow H \oplus (T$, *meuID*, $\sigma)$;
8: *Agende* Entregar(T) no tempo $T + \Delta$;
9: **fim para**
10: **fim enquanto**
11: **fim função**

teste é realizado para determinar se a mensagem já foi recebida, isto é, se ela já está em H. A primeira parte deste teste (se $T \in$dom(H)), verifica se já consta em H um determinado *timestamp* T, e a segunda parte (se $s \in$dom($H(T)$)) verifica se o processador s com *timestamp* T já consta no histórico (2.7). Ambos os testes são necessários porque podemos ter em H ao mesmo tempo um conjunto de mensagens tais como $(10, p1, \sigma_1)$, $(10, p2, \sigma_2)$ e $(11, p1, \sigma_3)$.

Por fim, assim como na tarefa de início, armazena-se no histórico H o fato que o processador s fez um *broadcast* da mensagem σ no tempo T (2.9). Por último, a entrega de mensagens com *timestamp* T é agendada para o tempo $T + \Delta$ (2.10).

Algoritmo 2 Procedimento para Receber

1: **função** TASK RECEIVE
2: **const** $\Delta = \delta + \epsilon$;
3: **var** U,T: Tempo; σ: Mensagem; s: Processador;
4: **enquanto** *ciclo* **faça** *receive*(T, s, σ) sobre c; U ←*clock*;
5: **se** $U \geq T + \Delta$ **então** *"mensagem atrasada"* (*vá para próximo ciclo*) **fim Se**
6: **se** $T \in$dom(H) & $s \in dom(H(T))$ **então** *"deja vu"* (*vá para próximo ciclo*) **fim se**
7: $H \leftarrow H \oplus (T$, *meuID*, $\sigma)$;
8: *Agende* Entregar(T) no tempo $T + \Delta$;
9: **fim enquanto**
10: **fim função**

A última tarefa, a de entrega, é utilizada pelos processadores para efetivamente entregar as atualizações aos processos. Em 3.3, a variável val recebe $H(T)$, isto é, os nomes dos processadores que iniciaram um *broadcast* atômico no tempo T. A seguir, veremos porque a variável val é necessária.

Enquanto existirem elementos em val (3.4), o processador p irá receber o processador com menor *timestamp* (para obedecer a propriedade de ordem) (3.5). Através de comandos $\overline{RECEIVE}$, os processos solicitam as atualizações ($val(p)$ representa a atualização transmitida por p no *timestamp* T sendo considerado) (3.6). O processo p cuja atualização foi entregue é excluído da variável val (3.7).

Quando todas as atualizações forem entregues, é realizada a operação de exclusão, representada por "$\backslash$", a qual apaga de H todos os *broadcasts* iniciados no tempo T. É por causa dessa forma de exclusão que precisamos da variável val.

Algoritmo 3 Procedimento para Entregar

1: **função** TASK DELIVER(T: Tempo)
2:     **var** p: Processador; val: Processador → Mensagem;
3:     $val \leftarrow H(T)$;
4:     **enquanto** dom(val) $\neq \{\}$ **faça**
5:         $p \leftarrow$ min(dom(val));
6:         $\overline{RECEIVE}$(val(p));
7:         $val \leftarrow val$ p;
8:     **fim enquanto**
9:     $H \leftarrow H$ T;
10: **fim função**

5.4.2 O protocolo tolerante à múltiplas falhas

Agora vamos abordar um caso mais complexo, onde múltiplas falhas podem ocorrer. As idéias expostas no protocolo tolerante a uma única falha deverão ser expandidas para que o protocolo continue funcionando corretamente. Um item a ser observado é que quando temos mais que uma falha, é necessário fazer adiantamento (este fato influenciará em muitas das alterações realizadas).

Anteriormente, vimos que para respeitar a propriedade de ordem, um processador deveria, além de entregar as mensagens por ordem de *timestamp*, esperar um determinado tempo para garantir que nenhuma mensagem com *timestamp* anterior ainda estava a caminho. Este tempo a ser esperado depende do maior atraso (pior caso) que pode ocorrer entre o momento que um processador inicia a transmissão de uma mensagem e o momento em que todos os processadores corretos a receberam.

Quando podem ocorrer $f > 1$ falhas, este pior caso depende de quantas vezes uma mensagem foi encaminhada antes de alcançar um processador correto. Para possibilitar identificar quantas vezes foi realizado o encaminhamento de uma determinada mensagem, as mensagens passam a ter a forma (T, s, σ, h), onde h é o *hop count*, cujo valor inicial é 1, e é incrementado a cada vez que a mensagem em questão é encaminhada.

Com $f > 1$ falhas, é possível termos a combinação de falhas de processador e de adaptadores, o que pode levar uma messagem a ter um atraso maior que δ. Com isso, além dos testes de "mensagem atrasada" e "*deja vu*", utilizados para o caso de uma única falha, é preciso determinar se uma mensagem é *timely*, ou seja, não tardia para encaminhamento. O novo teste de validade é dado por:

$$U < T + h(\delta + \epsilon)$$

Este teste garante que se um processador correto p aceita um *broadcast* (T, s, σ, h) que precisa ser encaminhado, então as mensagens (T, s, σ, $h + 1$) que p irá encaminhar também serão *timely* para todos os processadores corretos.

Vamos agora resolver o problema exposto no início desta seção, analisando os cenários que podem causar o pior atraso, para que possamos determinar o tempo de terminação. Considerando que o número de falhas é maior que um, a pior combinação de falhas que pode ocorrer é quando um processador falha após colocar a mensagem em apenas um *out-adapter* e então ocorre ou uma falha de desempenho no *out-adapter* ou uma falha de omissão no canal.

Caso o número de falhas seja par, isto é, $f = 2k$ para algum $k \geq 1$, $\Delta = (k+1)(\delta + \epsilon)$ é um tempo de terminação aceitável porque se p é um processador correto que aceita como *timely* uma mensagem (T, s, σ, h), vinda do maior canal c que necessita de adiantamento ($c < f+1-h$ com $h \leq k$), então p tem tempo suficiente para encaminhar (T, s, σ, $h+1$) para todos os processadores corretos antes que o tempo de entrega $T + \Delta$ ocorra em seu *clock*. O valor $k+1$ é devido ao cenário descrito no parágrafo anterior. Quando o número de falhas é par, isto é, $f = 2k,\ k \geq 1$, são necessários $(k+1)$ *hops* para que o primeiro processador correto seja alcançado.

O mesmo tempo de terminação é válido quando o cenário possui um número ímpar de falhas, ou seja, $f = 2k+1$. Embora um primeiro raciocínio leve a pensar que o tempo necessário seria de $\Delta = (k+\mathbf{2})(\delta+\epsilon)$, devemos observar que a última falha possível de ocorrer no cenário com número ímpar de falhas não impede que os processadores corretos recebam a mensagem. De forma mais detalhada, sempre que um processador correto p aceita uma mensagem (T, s, σ, k) em tempo local $T + k(\delta + \epsilon)$, mensagem essa que no cenário para número par de falhas poderia ser perdida por algum processo correto, o processador p irá encaminhar tal mensagem nos canais $k+1$, $k+2 = f+1-k$. Com isso, as últimas $(2k+1)$-ésimas falhas permitidas pela suposição 8 não irão conseguir impedir os demais processadores corretos de receber pelo menos uma cópia de (T, s, σ, $k+1$) em tempo $T + (k+1)(\delta+\epsilon)$.

Além de permitir a detecção de mensagens *non-timely*, o conhecimento do *hop count* também minimiza o número de canais em que uma mensagem precisa ser encaminhada. Para englobar esses novos tópicos, apresentamos uma nova definição para a *lazy forwarding rule*.

> *Lazy forwarding rule*: para iniciar um *broadcast*, um processo iniciador s enfileira mensagens (T, s, σ, 1) em seus *out-adapters* para os canais $1, 2, \ldots, f+1$ nesta ordem. Seja (T, s, σ, h), $h \leq k$ uma mensagem aceita por um processador $p \neq s$ e seja c o número identificador de canal mais alto pelo qual p recebe uma cópia da mensagem em tempo local $T + h(\delta+\epsilon)$. Se em $T + h(\delta+\epsilon)$ no *clock* de p, $c < f+1-h$, então p encaminha (T, s, σ, $h+1$) nos canais $c+1, \ldots, f+1-h$. Caso contrário, não é realizado encaminhamento.

A propriedade da unaminidade também é garantida: Se p é correto e aceita uma mensagem m em $T + \Delta$ sobre o *clock* de algum q então todo processo correto q também o faz. A prova desta propriedade pode ser encontrada em [4] .

Assim como no caso do protocolo de uma única falha, na ausência de falhas são enviadas apenas $f+1$ mensagens. No pior caso, $(f+1)+(f-1)(n-1) = (f-1)n+2$ mensagens são enviadas.

Uma nova tarefa é executada além das três mencionadas no protocolo de uma única falha (obviamente, com as devidas modificações). Tal tarefa é denominada *encaminhar*. Devido à similaridade com os algoritmos apresentados na seção anterior, apenas as modificações serão comentadas.

As modificações no procedimento para iniciar (algoritmo 4) se deram no tempo de terminação (4.2), pelos

Algoritmo 4 Procedimento para Iniciar

```
1: função TASK START
2:     const Δ = ⌊f/2(δ + ϵ) + (δ + ϵ);
3:     var T: Tempo; σ: Mensagem; s: Processador;
4:     enquanto ciclo faça SEND(σ); T ←clock;
5:         para c = 1 até f + 1 faça
6:             send(T, meuID, σ, 1) sobre c;
7:             H ← H ⊕ (T, meuID, σ);
8:             Agende Entregar(T) no tempo T + Δ;
9:         fim para
10:    fim enquanto
11: fim função
```

motivos já discutidos; os canais vão agora até $f+1$, sendo que antes considerávamos apenas 2 canais e a forma das mensagens incluiu o *hop count* h.

Algoritmo 5 Procedimento para Receber

```
1: função TASK RECEIVE
2:     const Δ = ⌊f/2(δ + ϵ) + (δ + ϵ);
3:     var U,T: Tempo; σ: Mensagem; s: Processador; h: Inteiro;
4:     enquanto ciclo faça receive(T, s, σ, h) sobre c; U ←clock;
5:         se U ≥ T + Δ então "mensagem atrasada" (vá para próximo ciclo) fim se
6:         se U ≥ T + h(δ + ϵ) então "tardia para encaminhamento" (vá para próximo ciclo) fim se
7:         se  T ∈ dom(H) & s ∈ dom(H(T)) então "deja vu" C(T)(s) ← max c,C(T)(s);
8:         senão H ← H ⊕ (T, meuID, σ);
9:             se h ≤ ⌊f/2 & c < f + 1 − h então
10:                C ← C ⊕ (T, s, c)
11:                Agende Encaminhar(T, s, h) no tempo T + h(δ + ϵ);
12:            fim se
13:        fim se
14:    fim enquanto
15: fim função
```

Na tarefa *Receber*, é introduzida a variável C, que também será utilizada no procedimento para encaminhar. Para cada *broadcast* (T, s) no histórico, a variável C grava o número identificador de canal mais alto no qual uma mensagem (T,s, *, *) foi recebida. Um determinado *timestamp* é eliminado de C no mesmo momento que é eliminado de H. Sendo assim, a tarefa de Entrega possui somente este descarte além do algoritmo para o caso de uma única falha (e por esse motivo não será escrita novamente).

Outras mudanças foram a inclusão do teste se uma mensagem é *timely* e um teste para verificar se é necessário realizar o encaminhamento da mensagem em questão.

Esta última tarefa é responsável por encaminhar uma determinada mensagem nos canais apropriados. Aqui

Algoritmo 6 Procedimento para Encaminhar

```
1: função TASK ENCAMINHAR(T: tempo; s: processador; h: inteiro)
2:     se C(T)(s) < f + 1 − h então
3:         para i = C(T)(s) + 1 até f + 1 − h faça
4:             send(T, s, H(T)(s), h + 1) em i
5:         fim para
6:     fim se
7: fim função
```

pode-se observar que os canais nos quais uma mensagem será enviada não são fixos, mas sim dependem do estado do sistema.

6 Algoritmo 2 – Bimodal Multicast ou Pbcast

6.1 Apresentação

O segundo algoritmo a ser estudado é o "Bimodal Multicast" de Kenneth Birman [2]. A maioria dos artigos anteriores a este se concentra em duas classes de problemas de *reliable multicast*. Na primeira, a concentração é o desempenho e a escalabilidade. A segunda enfatiza o rigor das definições de *reliability* que, em geral, incluem a atomicidade. O protocolo proposto por Birman permite obter tanto escalabilidade quanto confiança, que pode ser predita mesmo em condições altamente pertubadas.

A natureza crítica de diversas aplicações, como mercado de ações ou controle de tráfego aéreo, cria a necessidade de conhecer exatamente como tais sistemas se comportam nas condições esperadas, por parte dos desenvolvedores. Tais aplicações demandam um grande desempenho e escalabilidade. Em particular, elas possuem um volume consideravelmente alto de informações críticas para operação segura. No passado, esses sistemas eram identificados como *aplicações críticas de tempo real*, mas os computadores e redes atuais são tão rápidos que a verdadeira necessidade se tornou um *throughput estável*.

O *bimodal multicast* (ou *pbcast*) é uma boa escolha para esse propósito por vários motivos. Primeiro, a carga associada ao protocolo pode ser predita independentemente da escala sendo utilizada. Segundo, o protocolo tem seu foco num *throughput* estável. Assim, o projetista pode antecipar que um *bimodal multicast* irá consumir uma determinada porcentagem dos recursos de memória e de *bandwidth*.

6.2 Características do protocolo

O protocolo de *pbcast* satisfaz as seguintes propriedades:

- **Atomicidade**: o protocolo provê a "garantia de entrega bimodal". Ele garante uma grande probabilidade de que cada *multicast* chega em quase todos os processos, uma pequena probabilidade de chegar em poucos processos e uma probabilidade desprezível de chegar a uma quantidade intermediária de processos. Essa garantia pode ser expressa como "quase todos" ou "quase nenhum" processo recebe o *multicast*;

- **Estabilidade do throughput**: a variação esperada no *throughput* pode ser caracterizada e é pequena se comparada às taxas de *multicast* típicas.

- **Ordem**: as mensagens são entregues na ordem da FIFO (*First In First Out*), baseado em quem enviou.
- **Estabilidade do multicast**: o protocolo detecta a estabilidade das mensagens no sentido de que a atomicidade já foi obtida. Uma mensagem estável pode ser excluída (*garbage-collected*) com segurança e, se for desejado, o aplicativo pode ser informado.
- **Detecção de mensagens perdidas**: apesar de não ser freqüente em processos corretos, o modelo admite uma pequena probabilidade de que o *multicast* não chegue a alguns processos. Além disso, sabemos que a perda de mensagens é comum aos processos falhos. Se esse evento ocorrer, processos que não receberam mensagens são informados através do algoritmo de *anti-entropia* como será apresentado na Seção 6.4.2;
- **Escalabilidade**: os custos computacionais do protocolo (tempo e mensagens) são constantes ou possuem uma taxa de crescimento pequena em relação ao tamanho da rede.

Para fins de análise e simulação posterior, assumimos as mesmas premissas para as quais o protocolo foi proposto: o protocolo opera em uma rede na qual o *throughput* e a *reliability* ficam em torno de 75% das mensagens enviadas e que os *delays* dos *links* de rede são conhecidos. Além disso, também consideramos que os processos corretos respodem, a um evento de entrada, em um tempo limitado e conhecido (sincronização parcial). Novamente, isso corresponde a 75% dos processos da rede.

6.3 Estabilidade

Protocolos confiáveis de *multicast* – *broadcast* atômico em suas várias formas – sofrem uma interferência entre o controle de fluxo e os mecanismos de confiança. Isto pode causar um *throughput* instável quando a rede é aumentada, ou um comportamento errático. Na maioria das vezes, a garantia de estabilidade é extremamente importante: é necessário saber se o fluxo de dados é entregue de maneira estável e confiável. As Figuras 5 e 6 apresentam os protocolos virtualmente síncrono (tradicional) e o protocolo *pbcast* medidos em um processo perturbado e não perturbado. Na Figura 5, com 100 mensagens por segundo e na Figura 6, com 150 mensagens por segundo. Nota-se que o *pbcast* mantêm um *throughput* estável no *host* não perturbado. Além disso, no *host* perturbado o *pbcast* também é melhor do que o tradicional. A verificação de afirmações como essas é um dos objetivos ao propor a implementação de uma plataforma de simulação de *broadcasts*.

6.4 Descrição do protocolo

Pbcast é composto de dois subprotocolos. O primeiro é um método não confiável e hierárquico de *broadcast* que busca entregar as mensagens da *melhor forma possível*. Se o *IP multicast* está disponível, ele pode ser utilizado. O segundo é um protocolo *anti-entropia* de duas fases que executa uma série de rodadas não sincronizadas, onde durante cada rodada há uma primeira fase para detectar mensagens perdidas, e uma segunda para corrigir as perdas encontradas. Para facilitar a explicação, podemos assumir que os relógios estão sincronizados, o que faz com que as rodadas de *anti-entropia* aconteçam ao mesmo tempo para todos os processos e também que só ocorram entre um *broadcast* e outro. As duas afirmações não são requisitos para garantir o funcionamento do protocolo.

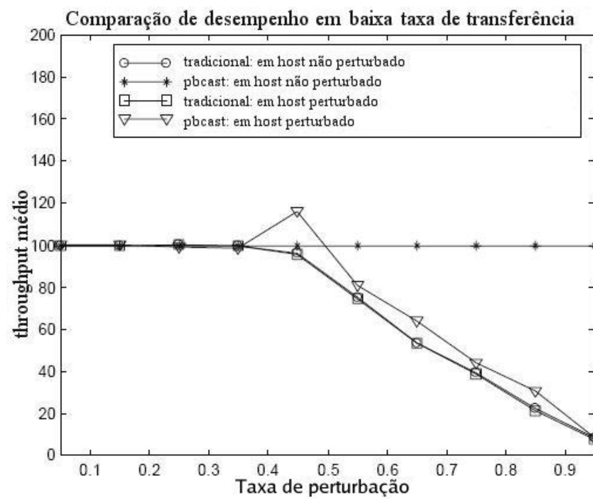

Figura 5: Protocolo virtualmente síncrono ("tradicional") *vs. pbcast* – 100 mensagens por segundo

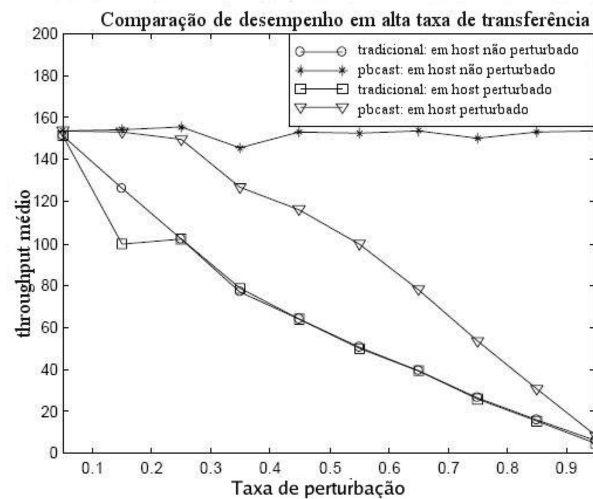

Figura 6: Protocolo virtualmente síncrono ("tradicional") *vs. pbcast* – 150 mensagens por segundo

6.4.1 Protocolo de disseminação

O primeiro estágio deste protocolo envia as mensagens usando uma primitiva de *multicast não confiável*, o que pode ser feito utilizando *IP multicast* ou, quando não disponível, usando um protocolo de disseminação randômica. No último caso, assumimos que os componentes estão todos conectados e sobrepomos uma *árvore geradora de multicast* "virtual" aos participantes. Cada processo tem uma variedade destas árvores pseudogeradas para o *broadcast* das mensagens, que é realizado anexando um identificador da árvore à mensagem antes do envio. Caso algum processo receba a mensagem, o protocolo de *anti-entropia* ainda garantirá a entrega de forma probabilísticamente confiável.

6.4.2 Protocolo anti-entropia de duas fases

O termo *anti-entropia* se refere ao protocolo [6] que detecta e corrige inconsistências em um sistema por *gossip contínuo*. No *gossip*, um membro encaminha as novas informações aos membros randomicamente escolhidos, combinando assim a eficiência da disseminação hierarquica com a robustez de protocolos de *flooding* (nos quais um membro envia as novas informações a todos os seus vizinhos).

No caso do nosso protocolo, em cada rodada um processo escolhe aleatoriamente outro de seu grupo e envia o seu histórico de mensagens. Quando este recebe um destes sumários e descobre que existe alguma mensagem que ainda não recebeu, ele envia um pedido ao disseminador daquela mensagem. Isto está ilustrado na Figura 7, onde as setas pontilhadas representam mensagens não recebidas.

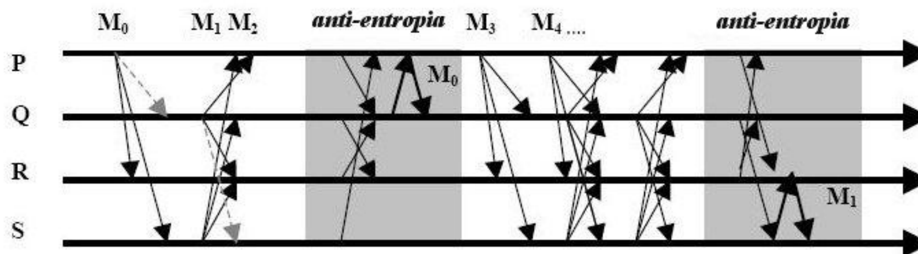

Figura 7: Protocolo *pbcast*

Ao contrário da maioria dos protocolos de *gossip*, este enfatiza a requisição das mensagens mais novas faltantes. Quando uma mensagem não recebida se torna muito antiga, ela é considerada como perdida. A vantagem desta estrutura é que evita cenários onde um processo sofreu problemas passageiros e não está conseguindo voltar ao mesmo estado do resto do grupo. Este problema poderia causar a sobrecarga dos *buffers* de outros processos diminuindo o desempenho do grupo como um todo. No entanto, mesmo as mensagens sendo marcadas como perdidas, o sistema pode ser configurado para que nos casos corretos esta perda seja muito rara, resultando apenas de casos em que o processo que iniciou o *broadcast* da mensagem parou.

Assim, no início do processo é definido um tempo a partir de parâmetros passados ao protocolo, sendo que tal valor é a duração de cada rodada. Em cada rodada, o protocolo de *anti-entropia* é executado novamente até um número máximo fixo de vezes. O produto desses parâmetros passados ao protocolo é denominado *fanout*. É possível também manter servidores de mensagens onde membros que acabaram de se conectar à rede ou aqueles que voltaram de um estágio de dormência poderiam requisitar uma seqüência de mensagens, possibilitando que estes alcancem o mesmo estado do resto do grupo.

Existem várias otimizações que podem ser feitas ao algoritmo de *anti-entropia* para limitar os custos em um cenário de falhas tais como:

1. Detecção de falha de software: se a resposta a um pedido de retransmissão está demorando mais do que uma rodada, provavelmente o processo é falho e o pedido é cancelado.

2. Limite de retransmissão na rodada: a quantidade de dados enviada em uma única rodada também é limitada. Ao atingir este limite, o processo simplesmente pára de transmitir dados. Isto impede que processos que se atrasaram devido a problemas e tentem pedir toda a informação atrasada de uma só vez.

3. Retransmissão cíclica: processos respondendo a pedidos de retransmissões analisarão as mensagens pedidas em rodadas anteriores, evitando que mensagens já enviadas e que podem estar no caminho sejam reenviadas.
4. Retransmissão do mais recente primeiro: os pedidos de retransmissão das mensagens mais recentes são atendidos primeiro. A estratégia de atender primeiro os pedidos de mensagens mais antigas pode causar numa queda de desempenho devido a processos com problema que não conseguem acompanhar os outros.
5. Numeração independente das rodadas: pode parecer que as rodadas avançam de maneira síncrona nos diferentes processos. No entanto, o protocolo proposto permite que cada um tenha sua própria numeração e funcione assíncronamente.
6. Grafos randômicos para escalabilidade: se pensarmos em utilizar esse protocolo em redes grandes, teremos dois problemas principais. O primeiro surge da necessidade que cada processo tem de ter uma lista de todos os membros para ser possível a realização do algoritmo de *anti-entropia*, o que causaria um fluxo muito grande de dados para manter as listas de membros de cada processo atualizada. O segundo problema enfrentado deve-se ao fato de que uma rede muito grande pode implicar a necessidade de valores muito grandes para o tempo de duração de cada rodada. Os dois problemas podem ser tratados fazendo com que cada participante só tenha conhecimento de um subgrupo dos membros. Assim, cada participante só recebe mensagens avisando da inclusão de um novo membro se este novo membro pertence ao seu subgrupo, e só envia mensagens para este subgrupo.
7. Multicast para algumas retransmissões: a utilização de multicast para a retransmissão de todas as mensagens implicaria em um *overhead* desnecessário. No entanto, se dentre os vários processos na implementação do artigo, dois pedirem a retransmissão de uma mensagem, a chance de outros também a terem perdido é grande, sendo assim justificável o multicast para esta retransmissão.

6.5 Comparações

Uma comparação entre os diferentes algoritmos apresentados é dificil de ser realizada. Enquanto o algoritmo de *broadcast* atômico é síncrono, o algoritmo de *multicast bimodal*, apesar de ter sido explicado em um funcionamento síncrono, não apresenta esta premissa como requisito e pode também ser utilizado de maneira assíncrona por processadores não sincronizados.

Outra característica importante que difere em ambos é a definição de *atomicidade*. No primeiro, a *atomicidade* é obtida quando todos ou nenhum dos processos corretos recebem a mensagem enviada. Por outro lado, no segundo, a *atomicidade* é definida da seguinte forma: uma dada mensagem será recebida por quase todos ou quase nenhum dos processos. Apesar destas diferenças, esperamos que seja possível uma comparação entre os algoritmos pois os dois foram utilizados para abordar um tipo comum de problema, o controle de tráfego aéreo. Ao simulá-los na plataforma proposta esperamos levantar suas principais semelhanças (se houver) e diferenças.

A utilização de algoritmos diferentes é uma característica importante do trabalho. Caso estes trabalhos fossem muito parecidos, uma análise comparativa das características exploradas seria possível sem necessidade de implementação da plataforma proposta. De fato, Birman [2] propôs o seu protocolo para permitir que

algoritmos desenvolvidos para trabalhar melhor em situações com um menor volume de informação possam ser implementados em conjunto com este.

Desta forma, propomos a implementação de três mecanismos diferentes a serem analisados na plataforma de simulação. O primeiro se trata do algoritmo de *broadcast atômico* proposto por F. Christian (Algoritmo 1, Seção 5). O segundo, será o algoritmo de *multicast bimodal* proposto por Birman (Algoritmo 2, Seção 6). Como terceiro algoritmo implementado na plataforma, dividiremos a rede em subconjuntos com um líder escolhido para cada subconjunto. Os líderes se comunicarão entre si através de uma implementação do primeiro algoritmo. Ao mesmo tempo, os líderes se comunicarão com os membros do subgrupo por meio do segundo algoritmo proposto. A quantidade de subgrupos no qual a rede será subdividida será um parâmetro a ser definido de forma a maximizar o desempenho.

Durante esta análise, será interessante verificar o desempenho das seguintes características:

- número de mensagens;
- tempo de terminação;
- número de processos informados;
- throughput;

A partir desta análise, espera-se ser possível classificar qual destas implementações é mais útil para diversos casos de aplicações diferentes.

7 Cronograma e plano de trabalho

Abaixo se encontra o cronograma de atividades a ser seguido.

Março de 2005 a Fevereiro de 2006												
Etapa	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez	Jan	Fev
1												
2												
3												
4												
5												
6												
7												
8												

Março de 2006 a Fevereiro de 2007												
Etapa	Mar	Abr	Mai	Jun	Jul	Ago[1]	Set	Out	Nov	Dez	Jan	Fev[2]
1	■	■	■	■	■	■	■	■	■	■	■	■
2	■	■	■	■	■	■	■	■	■	■	■	■
9	■	■	■	■	■	■	■	■				
10						■	■	■				
11							■	■				
12								■	■			
13									■	■		
14								■	■	■	■	■
15						■	■	■	■	■	■	■

1. **Coleta** de material bibliográfico.

2. **Desenvolvimento** do site da pesquisa.

3. **Estudo** das necessidades básicas da plataforma.

4. **Levantamento importância** da construção de uma plataforma com essa finalidade.

5. **Estudo do estado da arte** de técnicas *broadcast*.

6. **Comparação das vantagens e desvantagens** de tais técnicas.

7. **Estudo** das duas técnicas apresentadas no projeto de forma mais detalhada.

8. **Implementação da plataforma de simulação**

9. **Implementação das técnicas** previamente selecionadas..

10. **Proposição de melhorias** às técnicas estudadas bem como implementação de uma técnica própria.

11. **Testes de verificação** da implementação através da validação com as duas abordagens apresentadas nesta proposta.

12. **Documentação**.

13. **Escrita do relatório final**.

14. **Dissertação**.

15. **Escrita de artigos**. Escrita de artigos relacionados à pesquisa e submissão a eventos e/ou periódicos científicos relacionados ao tema.

Referências

[1] BACKES, M., AND CACHIN, C. Reliable broadcast in a computational hybrid model with byzantine faults, crashes and recoveries. In *Proceedings of International Conference on Dependable Systems and Networks* (2003), pp. 37–46. San Francisco, CA, USA.

[2] BIRMAN, K. P., HAYDEN, M., OZKASAP, O., XIAO, Z., BUDIU, M., AND MINSKY, Y. Bimodal multicast. In *ACM Transactions on Computer Systems* (1999), vol. 17, pp. 41–88.

[3] BRACHA, G. An asynchronous $\lfloor(n-1)/3\rfloor$-resilient consensus protocol. In *Proceedings of the 3rd annual ACM symposium on Principles of distributed computing* (1984), pp. 154–162.

[4] CRISTIAN, F. Synchronous atomic broadcast for redundant broadcast channels. In *The Journal of Real Time Systems, 2* (1990), Kluwer Academic Publishers, pp. 195–212.

[5] CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. Atomic broadcast: from simple message diffusion to byzantine agreement. In *Proccedings of the 15th International Symposium on Fault-Tolerant Computing* (1985), pp. 200–206.

[6] DEMERS, A., GREENE, D., HOUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. Epidemic algorithms for replicated database maintenance. In *ACM SIGOPS Operating Systems Review* (1988), vol. 22, pp. 8–32.

[7] FRØLUND, S., AND PEDONE, F. Revisiting reliable broadcast. In *Extended abstract* (2001), Hewlett-Packard Laboratories. Disponível em `www.hpl.hp.com/techreports/2001/HPL-2001-192.pdf`.