Protocolos para Difusão Confiável de Mensagens em Grupos de Comunicação

Aluízio Ferreira da Rocha Neto

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Protocolos para Difusão Confiável de Mensagens em Grupos de Comunicação

Aluízio Ferreira da Rocha Neto[1]

Agosto de 1997

Banca Examinadora:

- Prof. Dr. Ricardo de Oliveira Anido (Orientador)
- Prof. Dr. Orlando G. Loques Filho
  Depto. de Engenharia Elétrica - Universidade Federal Fluminense
- Prof. Dr. Edmundo Madeira
  Instituto de Computação - UNICAMP
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  Instituto de Computação - UNICAMP

[1]Apoio financeiro PICDT/UFRN - CAPES

Protocolos para Difusão Confiável de Mensagens em Grupos de Comunicação

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Aluízio Ferreira da Rocha Neto e aprovada pela Banca Examinadora.

Campinas, 26 de agosto de 1997.

Prof. Dr. Ricardo de Oliveira Anido
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 22 de agosto de 1997 pela Banca Examinadora composta pelos Professores Doutores

Prof. Dr. Orlando Gomes Loques Filho

Prof. Dr. Edmundo Roberto Mauro Madeira

Prof. Dr. Ricardo de Oliveira Anido

À minha namorada Bartira pelo incentivo e amor dispensados à distância.

Aos meus pais e irmãos pela compreensão e apoio durante esta árdua caminhada.

Agradecimentos

Ao meu orientador, Ricardo Anido, por ter me apresentado o tema da dissertação e pela amizade sincera com que me auxiliou durante todo o curso.

Aos meus amigos de república, Henrique Eduardo e Fátima Vitória, por terem compartilhado comigo os momentos de alegrias e tristezas durante este período.

Aos amigos de Natal, que também aqui estiveram, entre eles: Hugo Alexandre, Agostinho Junior, Guilherme Queiroz, Marco Antônio e João Matos, pelo "clima nordestino" que criaram, amenizando a saudade da nossa terra.

Aos professores da Unicamp e aos colegas de curso, por terem me propiciado um ambiente de pesquisa bastante rico e produtivo, e pela troca de conhecimentos.

E finalmente à UFRN e à CAPES pelo apoio financeiro, sem o qual não seria possível realizar este trabalho.

O meu muito obrigado.

Resumo

Em um Sistema Distribuído, computadores cooperam entre si para fornecerem um serviço. A troca de mensagens entre estes computadores através dos canais de comunicação é uma das características destes sistemas. Muitas aplicações distribuídas, como por exemplo sistemas de banco de dados replicados, envolvem comunicação freqüente entre os participantes do sistema. Nestes casos, uma abstração muito útil é a **Comunicação em Grupo**. Neste tipo de comunicação, um computador envia uma mensagem destinada a um grupo de outros computadores encarregados de fornecerem um serviço distribuído.

Alguns modelos de comunicação em grupo têm sido propostos e utilizados em ambientes com aplicações distribuídas. Dentre estes modelos destaca-se o modelo de *Difusão Confiável*. Caracteriza-se uma difusão como confiável quando a mensagem difundida ou é recebida por todos os membros do grupo ou não é recebida por nenhum deles. Além disso, a difusão mantém uma ordem consistente das mensagens trocadas dentro do grupo de comunicação.

Esta dissertação apresenta um estudo sobre os protocolos que implementam difusões confiáveis para grupos de comunicação em sistemas distribuídos. É descrito e analisado um conjunto de protocolos de comunicação em grupo presentes na literatura. Uma outra contribuição deste trabalho é a proposta de um novo protocolo de difusão confiável.

Abstract

In a Distributed System, computers cooperate to provide a service. One of the features in such systems is the exchange of messages among theses computers using a communication network. Many distributed applications, like replicated databases, require frequent broadcasts to/from the servers of data. In this case, a useful abstraction is the **Group Communication**: a computer sends a message to a group of other computers which provide a distributed service.

Some models of group communication were developed and used in distributed applications. One of the most important models introduced to provide group communication was the *Reliable Broadcast*. This model ensures that messages sent to the group are delivered to all members of the group or are not delivered to any of them. Moreover, the broadcast keep a consistent order of the messages exchanged within the group.

This thesis is a survey on reliable broadcast protocols for group communication in distributed systems. It describes and analyzes the models of communications based on data and control structures utilized in the protocol. Another contribution of this thesis is the proposal of a new reliable broadcast protocol.

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

Sistemas Distribuídos fornecem alta disponibilidade de informações e excelente desempenho a um baixo custo. As tarefas de uma aplicação podem ser divididas entre vários computadores e os dados podem ser replicados e compartilhados. Entretanto, devido às dificuldades de se coordenar tarefas e de se compartilhar recursos entre computadores, a potencialidade prometida por Sistemas Distribuídos não é freqüentemente conseguida. É sensivelmente mais complexo desenvolver softwares distribuídos do que softwares centralizados, visto que atingir uma consistência global do sistema é, normalmente, uma tarefa bastante árdua. Um dos principais motivos que dificultam o desenvolvimento de aplicações distribuídas é a ocorrência de falhas aleatórias nos computadores e canais de comunicação do sistema. Espera-se que as aplicações continuem executando apesar da ocorrência de falhas parciais.

Tipicamente em um Sistema Distribuído, um número de computadores cooperam entre si para fornecerem um determinado serviço. O principal mecanismo utilizado em tal sistema é a troca de mensagens entre estes computadores através dos canais de comunicação. Muitas aplicações distribuídas (como sistemas de banco de dados replicados, *groupware*, etc.) envolvem comunicação freqüente entre os participantes do sistema. Nestes casos, uma abstração muito útil é a **Comunicação em Grupo**. Neste tipo de comunicação, um processo [1] envia uma mensagem destinada a um grupo de outros processos encarregados de fornecerem um serviço potencialmente distribuído.

Alguns modelos têm sido propostos e utilizados em ambientes com aplicações distribuídas. Dentre estes modelos destaca-se o modelo de *Difusão Confiável*. Caracteriza-se uma difusão como confiável quando esta garante a entrega e mantém uma ordem consistente das mensagens para todos os membros do grupo de comunicação.

Vários protocolos para difusão confiável foram desenvolvidos, cada um utilizando um algoritmo diferente, determinado, normalmente, pelo tipo de aplicação e topologia de rede nas quais o protocolo será utilizado. Isto fez com que a escolha de um protocolo de difusão, para se implementar a comunicação em grupo em um Sistema Distribuído, não seja uma tarefa trivial, em razão também da falta de literatura na área apresentando uma análise comparativa destes

[1] Utilizaremos o termo processo como sendo uma unidade lógica de processamento dentro de um sistema.

protocolos.

Este trabalho tem como objetivo o estudo da difusão confiável e dos protocolos que a implementam, bem como a apresentação de um novo algoritmo para um protocolo de difusão confiável de mensagens em um grupo de comunicação.

1.1 Confiabilidade em Sistemas Distribuídos

Define-se Sistemas Distribuídos como uma coleção de computadores autônomos que comunicam-se por troca de mensagens através de uma rede de comunicação, e com software projetado para produzir uma facilidade de computação integrada [DK94].

As aplicações dos sistemas distribuídos variam de serviços de propósito geral para grupos de usuários a sistemas de automação bancária e comunicação multimídia.

As principais características dos sistemas distribuídos são: suporte para compartilhamento de recursos, abertura, concorrência, escalabilidade, confiabilidade e transparência. Confiabilidade é uma característica muito importante e ao mesmo tempo difícil de ser conseguida, visto que ela implica em uma série de propriedades desejáveis em um sistema distribuído, tais como:

- *Tolerância a Falhas*: O sistema permanece correto apesar da ocorrência de falhas.
- *Disponibilidade alta ou contínua*: A disponibilidade de informações está intimamente relacionada com tolerância a falhas, isto é, os dados são fornecidos mesmo depois de falhas.
- *Desempenho*: Espera-se que o tempo de resposta do sistema seja aceitável ou o menor possível.
- *Recuperabilidade*: Os componentes que falharam podem reiniciar.
- *Consistência*: As ações dos múltiplos componentes do sistema devem estar coordenadas, de forma que podemos observá-los como uma entidade única.
- *Segurança*: Autentica o acesso aos recursos e serviços.
- *Privacidade*: Protege a identidade e a localização dos objetos do sistema (usuários, dados, etc.).

A maioria destas propriedades, excluindo Segurança e Privacidade, são derivadas da capacidade do sistema reagir a ocorrência de falhas, ou seja, ser tolerante a falhas. As aplicações em um sistema distribuído, compostas de vários processos em diferentes computadores, que utilizam difusão de mensagens estão sujeitos aos seguintes tipos de falhas [Ros93, DB84]:

- **Falha de parada**: O processo apenas pára de funcionar, neste caso, pára de enviar ou receber mensagens.

- **Falha de omissão:** O processo omite o envio de algumas mensagens prescritas pelo protocolo. Em sistemas síncronos, este tipo de falha pode ser causada devido a atrasos na transmissão das mensagens.
- **Falha de rede:** O componente de ligação do computador ao canal da rede pode quebrar e isolar o processo do resto do sistema. Neste caso, pode-se considerar que houve uma falha de processo.
- **Falha de particionamento de rede:** O canal da rede pode se fragmentar e dar origem a duas ou mais sub-redes.
- **Falha de temporização:** O processo continua funcionando e gera resultados corretos, porém não no prazo esperado. As falhas de temporização podem ser ainda classificadas em *falhas de atraso* e *falhas por antecipação.*
- **Falha de valor ou falha bizantina:** O processo continua funcionando após a falha, porém passa a exibir um comportamento arbitrário. Tal processo pode reter mensagens que eram para ser enviadas ou enviar mensagens não prescritas pelo protocolo, e até mesmo cooperar com outro processo (falho) para interromperem o sistema.
- **Perda de mensagens:** Uma mensagem enviada por um processo não chega a seu destino por alguma falha no canal de comunicação. Um atraso muito grande na transmissão de uma mensagem pode induzir um processo a pensar que houve uma perda de mensagem.

Um sistema é considerado tolerante a falhas se possuir pelo menos uma das seguintes propriedades:

- Ter um comportamento bem definido após a falha de seus componentes; ou,
- Continuar fornecendo seus serviços mesmo após as falhas de alguns componentes. Neste caso, diz-se que o sistema *mascara* as falhas ocorridas.

Existem diversos mecanismos de software e de hardware que podem ser utilizados com o objetivo de tornar um sistema tolerante a falhas. Este texto trata particularmente da difusão confiável de mensagens – uma ferramenta de programação muito útil para a construção de sistemas tolerantes a falhas.

1.2 Comunicação em Grupo e Mensagens de Difusão

A fim de tratar da crescente complexidade das aplicações, uma série de conceitos foram introduzidos. O conceito de *processo* foi definido para se ter a noção de atividade seqüencial de um programa. Conceitos como portas, canais, caixas de mensagens, etc., foram utilizados para solucionar problemas de comunicação. Um dos últimos conceitos introduzidos foi o de **grupo** [BJ87]; um grupo é um conjunto de processos que implementam algum serviço: no nível

da aplicação um cliente deste serviço o enxerga apenas como um serviço abstrato e não como processos individuais que o implementam.

Para a comunicação com um grupo de processos, foi criado o conceito de **mensagem de difusão**. Esta é uma mensagem que é enviada por um processo qualquer para os membros de um grupo de processos. Mensagens de difusão fornecem uma infra-estrutura significativa para se conseguir tolerância a falhas em aplicações distribuídas [DK94].

Há muitas maneiras de se implementar a difusão de uma mensagem, atendendo-se a determinados requisitos. A mais simples delas é a difusão não-confiável, a qual não fornece nenhuma garantia sobre a entrega ou ordenação das mensagens enviadas para o grupo. Este tipo de difusão é implementada na maioria dos sistemas operacionais atuais, como por exemplo, no UNIX através do protocolo de *Multicasting* fornecido pela família de protocolos TCP/IP [Ste94].

1.2.1 Utilizando mensagens de difusão

Mensagens de difusão são uma ferramenta bastante útil para a construção de sistemas distribuídos com, por exemplo, as seguintes características [DK94]:

1. *Tolerância a falhas baseada em serviços replicados*: Um serviço replicado consiste de um grupo de servidores. Os pedidos dos clientes são difusões para todos os membros do grupo, e cada membro realiza uma operação idêntica. Mesmo quando alguns dos membros falham, os clientes ainda podem ser servidos.

2. *Localização de objetos em serviços distribuídos*: Mensagens de difusão podem ser usadas para localizar objetos dentro de um serviço distribuído, tais como um arquivo dentro de um sistema de arquivo distribuído.

3. *Melhor desempenho através de dados replicados*: Dados são replicados para se diminuir o tempo de resposta a um pedido de informação, melhorando a sua disponibilidade e aumentando o desempenho do serviço. Toda vez que um dado muda, o novo valor é difundido para os processos que gerenciam as réplicas.

4. *Múltiplas comunicações*: Difusão para um grupo pode ser usada para notificar os processos quando algum evento acontece, por exemplo, um sistema de *grupo de notícias* (*newsgroup*) pode notificar os usuários interessados quando uma nova mensagem é postada em um grupo particular.

A partir destes modelos de comunicação em grupo podemos derivar algumas propriedades importantes para protocolos de difusão de mensagens.

Atomicidade

No caso 1 acima é necessário que cada servidor receba todos os pedidos para que todos os servidores tenham executado a mesma tarefa mantendo os seus estados consistentes entre si.

Isto requer uma difusão atômica[2]:

Difusão Atômica : Uma mensagem transmitida por difusão atômica ou é recebida por todos os membros do grupo ou não é recebida por nenhum deles. Nós podemos considerar os processos falhos como não pertencentes a qualquer grupo.

Neste modelo de difusão, é possível evitar que uma determinada ação seja tomada quando um dos membros do grupo não recebeu a mensagem, o que pode ocasionar inconsistências. Esta abordagem é muito utilizada em Sistemas Distribuídos com transações críticas, como, por exemplo, controle de acesso a um determinado recurso compartilhado por vários processos. Entretanto, há aplicações que não necessitam apenas da garantia de entrega a todos do grupo, elas exigem também que todos recebam as mensagens de diferentes fontes em uma mesma ordem, como explicitado no caso 3.

Difusão Confiável : Garante difusão atômica com uma ordenação das mensagens conhecida por todos os membros do grupo.

A difusão confiável, além de garantir que todos os membros do grupo recebem as mensagens difundidas, possibilita que se defina a ordem com que estas mensagens são entregues ao grupo.

Ordenação de Mensagens

Há 4 abordagens para a ordenação de mensagens de difusão [SS91]:

- *Nenhuma ordenação*: Deixa a cargo dos desenvolvedores das aplicações a difícil tarefa de manter a consistência na entrega das mensagens.
- *Ordenação FIFO (First In, First Out)*: "*O primeiro a chegar é o primeiro a sair*". Isto é, as mensagens de uma única fonte são entregues na ordem em que elas foram enviadas por este membro.
- *Ordenação causal*: Garante que todas as mensagens que são relacionadas entre si são ordenadas e entregues obedecendo estas relações. Em resumo seria o seguinte: se as mensagens estão em ordem FIFO e se um membro depois de receber uma mensagem A envia a mensagem B, é garantido que todos os membros receberão A antes de B.
- *Ordenação total*: Cada membro recebe todas as mensagens em uma mesma ordem prescrita pelo protocolo. Esta ordenação é mais forte que qualquer das outras e torna a programação mais fácil, mas é mais difícil de implementar.

Para ilustrar a diferença entre ordenação FIFO e total, considere um serviço que armazena registros para processos clientes. Assuma que o serviço replica os registros em cada servidor para aumentar a disponibilidade e confiabilidade, e que este serviço quer garantir que todas

[2] A expressão atômica significa inquebrável, indivisível e, neste caso, de apenas um estado.

as réplicas estejam consistentes. Se um cliente pode apenas atualizar seus próprios registros, então é necessário que todas as mensagens de um mesmo cliente sejam ordenadas. Neste caso, a ordenação FIFO pode ser usada. Entretanto, se um cliente pode atualizar qualquer registro, então ordenação FIFO não é suficiente. Uma ordenação total nas atualizações é necessária para assegurar a consistência entre as réplicas. Para observar isto, assuma que dois clientes, C_1 e C_2, enviam uma atualização para o registro X ao mesmo tempo. Como estas duas atualizações estarão totalmente ordenadas, ou (1) todos os servidores recebem primeiro a atualização de C_1 e depois a de C_2, ou (2) recebem primeiro a atualização de C_2 e então a de C_1. Em qualquer caso, as réplicas permanecerão consistentes, pois todos os servidores aplicam as atualizações na mesma ordem. Ordenação causal também não seria suficiente, pois como C_1 não tem relação com C_2, os servidores poderiam aplicar as atualizações em ordens diferentes, resultando em inconsistências entre as réplicas.

Estes conceitos de ordenação de mensagens serão detalhados e melhor explicados no Capítulo 2.

1.2.2 Semânticas da Comunicação em Grupo

É muito importante para um serviço de comunicação em grupo manter uma semântica bem definida para tal serviço. O desenvolvedor da aplicação pode confiar nestas semânticas quando projetar aplicações que usam tais serviços de comunicação em grupo. As semânticas devem especificar tanto os compromissos quanto as garantias fornecidas pelo ambiente de programação distribuída que implementam tal comunicação em grupo.

O sistema ISIS definiu e mantém as semânticas de *sincronismo virtual* [BJ87]. Sincronismo virtual assegura que todos os processos pertencentes a um grupo acompanham as mudanças na configuração do grupo, quando estas ocorrem, no mesmo tempo lógico. Além disso, todos os processos pertencentes a uma configuração do grupo entregam o mesmo conjunto de mensagens para esta configuração. É garantido, ainda, que uma mensagem é entregue na mesma configuração em que ela foi difundida.

Sincronismo virtual assume falhas dos tipos: perdas de mensagens e paradas de processos. Um processo que pára pode nunca (ou não é permitido) se recuperar. Quando ocorre um particionamento[3] da rede, o sincronismo virtual assegura que os processos no segmento contendo a maioria dos processos, o segmento primário, são capazes de progredir; os processo nos outros segmentos são bloqueados.

Para superar algumas fraquezas no modelo inicial de sincronismo virtual introduzido no Sistema ISIS, a equipe de desenvolvimento do Sistema Totem definiu o *sincronismo virtual estendida* [Aga94, BLP96, Ami95]. Este conceito será melhor detalhado no Capítulo 2.

[3] Um particionamento de rede significa que ocorreu uma falha no canal de comunicação resultando na divisão da rede em dois ou mais segmentos, permitindo que os processos de um mesmo segmento ainda se comuniquem entre si.

1.3 Contribuições da Dissertação

Este trabalho se constitui em um estudo sobre os protocolos que implementam difusões confiáveis para grupos de comunicação em sistemas distribuídos. São observados, para cada protocolo, as estruturas de dados e controle utilizados na implementação e alguns aspectos de tolerância a falhas. É descrito e analisado em uma forma comum um conjunto de protocolos de comunicação em grupo presentes na literatura. Dentre estes protocolos, podemos citar: Chang e Maxemchuk [CM84], ISIS/HORUS [BJ87, SS91, BM96, HK95], Newtop [ES93, ES94], Trans/Total [PMMSA90], Transis [YAM92a, YAM92b], Psync [LLPS89] e Totem [AC93, Aga94, AC95].

O critério adotado para descrever os protocolos baseia-se nos seguintes itens: 1) classificação do protocolo, com base na estrutura de controle utilizada para a difusão; 2) forma de difusão de uma mensagem; 3) comportamento do protocolo na ocorrência de falhas; 4) demonstração da correção; 5) desempenho médio, com base no número de mensagens geradas ou tempo necessário para uma difusão; e 6) exemplo de operação do protocolo.

Para alguns protocolos descritos neste trabalho não foi possível a apresentação de todos os itens citados acima, devido a dificuldades em se encontrar material que apresente todos os aspectos do protocolo. Assim, como principal contribuição deste trabalho pode-se citar a análise do número de mensagens enviadas por difusão para todos os a classificação dada para cada protocolo.

Ainda como contribuição da dissertação, destaca-se a proposta de um novo protocolo de difusão confiável. O protocolo proposto, apresentado no Capítulo 7, combina as estratégias de uso de ficha, baseando-se no modelo de Chang e Maxemchuk [CM84], com a de Grafo de Contexto, introduzida pelo protocolo Psync [LLPS89].

Capítulo 2

Sistemas Distribuídos e Protocolos de Difusão

Os componentes de um sistema distribuído estão tanto lógica como fisicamente separados; eles devem se comunicar para que possam interagir. Sistemas e aplicações distribuídas são compostas de componentes de software separados que interagem a fim de realizarem tarefas. Nós podemos assumir que todos os componentes que requerem ou fornecem acesso a recursos em sistemas distribuídos são implementados como processos [DK94].

A comunicação entre um par de processos envolve operações de envio e recebimento de mensagens que juntas resultam em:

a) *transferência de dados* do ambiente do processo remetente para o ambiente do processo receptor;

b) algumas operações de comunicação, como a *sincronização* da atividade de recebimento com a atividade de envio de forma que um dos dois processos não prossiga na execução até que o outro realize uma ação que o libere.

Para (a) ocorrer, os processos da comunicação devem compartilhar um canal de comunicação — um meio para que os dados sejam transferidos entre eles, enquanto que (b) está implícito na operação de todas as primitivas de programação para comunicação em grupo.

Os mecanismos básicos de programação estão na forma de duas primitivas principais: *envia* e *recebe*. Estas primitivas realizam ações de **passagem de mensagem** entre um par ou um grupo de processos. Cada ação de passagem de mensagem envolve a transmissão pelo processo remetente de um conjunto de dados (uma mensagem) através de um modelo de comunicação específico (um canal ou porta) e a aceitação pelo processo receptor desta mensagem. O mecanismo pode ser **síncrono** ou *bloqueante*, significando que o remetente espera depois de transmitir uma mensagem até que o receptor tenha recebido esta mensagem, ou ele pode ser **assíncrono** ou *não-bloqueante*, o que significa que a mensagem é posta em uma fila de mensagens esperando que o receptor a processe e, enquanto isso, o remetente é liberado para realizar outras tarefas. A

operação *recebe* normalmente bloqueia o processo receptor quando não há mensagens disponíveis nesta fila.

Sistemas distribuídos podem ser projetados completamente em termos de *passagem de mensagens*, mas há alguns padrões de comunicação bastante úteis e essenciais para o projeto e construção de aplicações distribuídas. Dentre estes, estão o modelo **cliente-servidor** para a comunicação entre pares de processos e o modelo **difusão em grupo** para a comunicação entre grupos de processos cooperativos.

Comunicação cliente-servidor: Esta comunicação é orientada para o provimento de serviços. A utilização de um serviço consiste de:

1. transmissão de um pedido por um processo cliente para um processo servidor;
2. execução do pedido pelo servidor;
3. transmissão de uma resposta pelo servidor para o cliente.

Este padrão de comunicação cliente-servidor pode ser implementado em termos de operações básicas de passagem de mensagens e é comumente apresentado no nível de linguagens de programação como uma *Chamada de Procedimento Remoto* (RPC), que consiste de operações similares a abstração de chamada de procedimentos em uma linguagem e é implementada em um *protocolo de pedido-resposta* [DK94, Capítulo 5].

Comunicação por difusão em grupo: Em um padrão de comunicação por difusão em grupo, os processos também interagem por passagem de mensagens, mas neste caso o destino de uma mensagem não é um simples processo, mas um grupo de processos. Para uma operação simples de *envia* ao grupo corresponde a uma operação de *recebe* realizada por cada processo membro do grupo, como ilustrado pela Figura 2.1. O envio de uma mensagem para os membros de um determinado grupo de comunicação é conhecido como **Multicasting**. O termo **broadcasting**, comum na literatura, é utilizado nesta dissertação para se referir à capacidade de difusão por algumas tecnologias de rede, como a *Ethernet* [Ste94]. Assim, um *broadcast* é o envio de uma mensagem para todos os computadores da rede. Um *multicast* pode fazer uso da difusão por hardware mas isto representa apenas uma possível estratégia de implementação.

Pode haver ou não suporte de hardware para a comunicação em grupo no níveis de rede. Sem este suporte, uma mensagem de difusão tem que ser enviada seqüencialmente pelo protocolo de comunicação aos membros de um grupo de processos; ao passo que com este suporte, a mensagem pode ser enviada em paralelo. Portanto, a comunicação com grupo de processos é independente da existência de suporte de hardware para difusão.

2.1 Arquitetura do Sistema

A Figura 2.2 ilustra uma possível arquitetura de software e hardware existente em um sistema distribuído. Nesta figura uma linha horizontal significa que os serviços fornecidos pela caixa

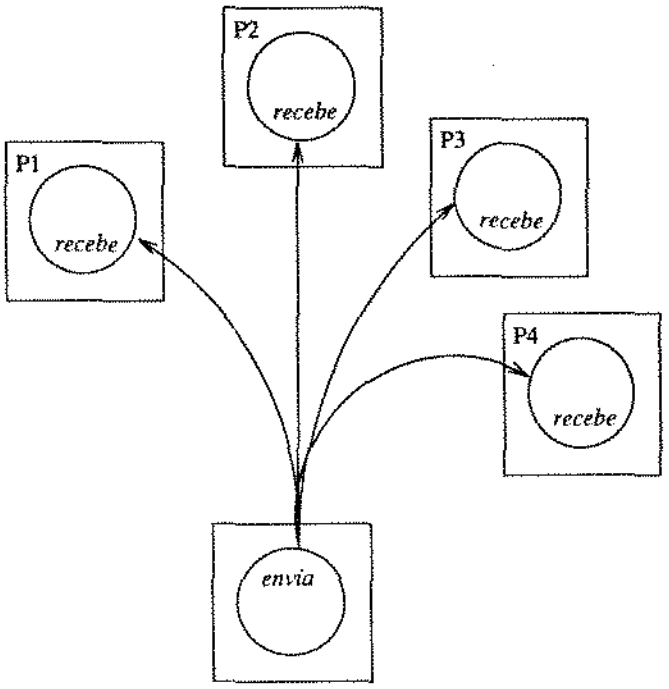


Figura 2.1: Difusão para um grupo de processos.

abaixo da linha são diretamente utilizados pelos componentes acima dela.

Os principais componentes para a concepção de um sistema distribuído são os *serviços abertos* e os *ambientes para programação distribuída*. Serviços abertos são recursos de software e hardware implementados na maioria dos ambientes computacionais com o intuito de fornecerem os requisitos básicos para o suporte a um sistema distribuído. Por exemplo, para que um sistema distribuído composto de plataformas heterogêneas, como microcomputadores e estações de trabalho, pudesse funcionar foi necessário o desenvolvimento de software e hardware de comunicação comuns às diversas plataformas, como a família de protocolos TCP/IP utilizando a tecnologia de rede *Ethernet*.

O ambiente de programação distribuída facilita a construção dos sistemas distribuídos pois fornece ao desenvolvedor uma linguagem com suporte a todas as ferramentas de comunicação. Tais linguagens fornecem protocolos para comunicação em grupo. Dentre estes protocolos, destaca-se o protocolo para difusão confiável de mensagens, abordado nesta dissertação.

Alguns destes ambientes, abordados neste trabalho, são: o ISIS/HORUS [BJ87, SS91, BM96, HK95], o Transis [YAM92a, YAM92b] e o Totem [AC93, Aga94, AC95]. O protocolo de difusão confiável de Chang e Maxemchuk [CM84] descrito no Capítulo 3 não faz parte de um ambiente de programação distribuída e, assim, pode estar implementado no sistema como um serviço aberto.

São os ambientes de programação distribuída e os serviços abertos que implementam, normalmente, a comunicação em grupo nos sistemas distribuídos, embora já existam sistemas operacionais com tal facilidade, os chamados Sistemas Operacionais Distribuídos. Um exemplo disto

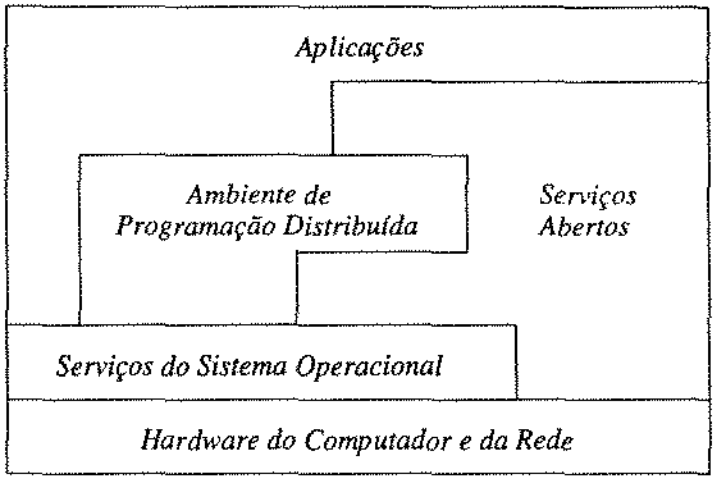


Figura 2.2: Arquitetura de um Sistema Distribuído.

é o Sistema Operacional Amoeba [KT91, KT92], idealizado por A. S. Tanembaum. Neste, existe uma camada de comunicação em grupo que implementa difusão confiável para um conjunto de maquinas de um sistema distribuído [HB89].

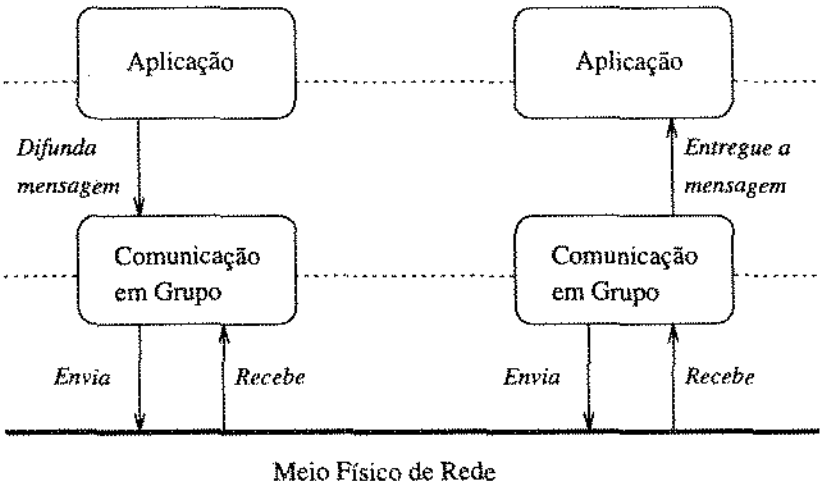


Figura 2.3: Arquitetura de uma comunicação em grupo.

Esta camada de comunicação em grupo faz acesso ao meio físico de rede e tem uma interface direta com a aplicação distribuída, como pode ser exemplificado na Figura 2.3. Um dos aspectos importantes nesta camada é que nela existe uma diferença entre receber uma mensagem e entregar esta mensagem à aplicação. Assim, para uma mensagem já recebida, a sua entrega pode ser retardada a fim de que o protocolo de difusão satisfaça os requisitos de ordenação e recebimento por todos do grupo.

Para prover os mecanismos de comunicação em grupo, esta camada se utiliza dos protocolos de difusão confiável discutidos na seção seguinte.

2.2 Protocolos de Difusão Confiável

Muitas aplicações que executam em um sistema distribuído requerem o compartilhamento de informações entre vários processos, assim como a sincronização das ações destes processos. Um exemplo típico de tal aplicação são os *Sistemas de Banco de Dados Distribuídos*, que replicam os dados em diferentes pontos de uma rede a fim de proverem uma maior disponibilidade destes dados para os seus clientes. Ferramentas úteis para compartilhar informações e manter dados replicados são os *protocolos de difusão confiável* [Sch88]. Tais protocolos propagam informação de um processo para um grupo de processos de tal forma que todos os destinos operacionais recebem estas informações apesar da ocorrência de falhas no sistema. Esta propriedade é chamada de *entrega confiável de mensagem*. Além disto, um protocolo de difusão confiável fornece uma forma de *ordenação das mensagens*. Portanto, este tipo de protocolo garante que todas as mensagens são recebidas em uma mesma ordem por todos os membros do grupo de comunicação.

Existe um ponto de discussão que trata do quanto de ordenação um protocolo fornece e quanto de atraso por sincronização é necessário para implementar esta ordenação. Uma difusão FIFO, por exemplo, pode ser implementada eficientemente nos canais de comunicação existentes adicionando apenas um *número de seqüência* em cada mensagem. Uma difusão com ordenação total, por outro lado, é muito mais caro de implementar nos sistemas estudados neste trabalho. Esta última forma de difusão requer duas ou mais fases de troca de mensagens entre os processos antes que uma mensagem possa ser entregue à aplicação.

2.3 Ordenação de Mensagens

Esta seção seguinte apresenta diferentes propriedades de ordenação e descreve como tais propriedades podem ser implementadas em um sistema completamente seguro no qual os processos não falham. Na seção 2.4 nós examinaremos como os diferentes tipos de protocolos podem *mascarar* as falhas.

2.3.1 Difusão Não-ordenada

A forma mais simples de difundir uma mensagem é simplesmente enviar uma cópia desta mensagem para todos os processos destinos individualmente, ou em paralelo por *broadcast*. A Figura 2.4 ilustra isto. Ela mostra um sistema com quatro processos pertencentes a um grupo de comunicação. O tempo progride da esquerda para a direita, e as setas diagonais representam o envio de mensagens. A Figura mostra o processo p_1 difundindo duas mensagens (a e b) para p_2, p_3 e p_4. As duas mensagens chegam na mesma ordem em p_2 e p_3, mas p_4 as recebe em uma ordem diferente. Pelo fato de que esta forma de difusão não garante qualquer ordem específica de entrega, nós a chamamos de uma *difusão não-ordenada*.

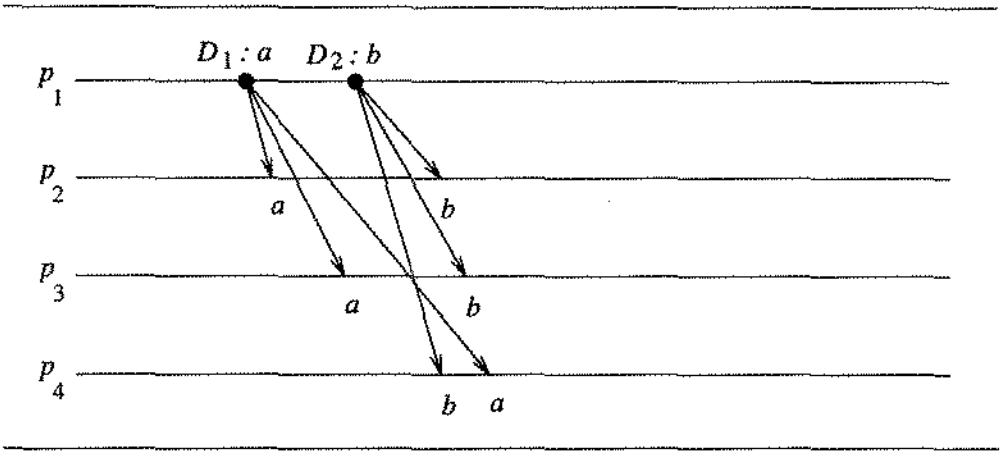


Figura 2.4: Difusão não-ordenada.

2.3.2 Ordenação FIFO

Se o canal de comunicação possibilita um envio de mensagens FIFO (*First In, First Out*), isto é, não há ultrapassagens nas transmissões das mensagens, então a difusão não-ordenada satisfaz uma propriedade de ordenação um pouco mais forte: toda difusão de mensagens pelo mesmo processo será entregue, em qualquer destino, na mesma ordem em que as mensagens foram enviadas. Mesmo se o canal não fornece transmissões FIFO, não é difícil implementar ordenação FIFO. Basta cada processo adicionar um número de seqüência para cada mensagem. Nós chamamos isto de difusão FIFO. Na Figura 2.5, por exemplo, o processo p_1 difunde duas mensagens, primeiro a e depois b. Os processos p_3 e p_4 recebem estas mensagens na ordem em que elas foram enviadas. As difusões D_2 e D_3, entretanto, são enviadas por diferentes processos; tais difusões podem ser entregues em diferentes ordens em diferentes nós do grupo, como mostrado no exemplo.

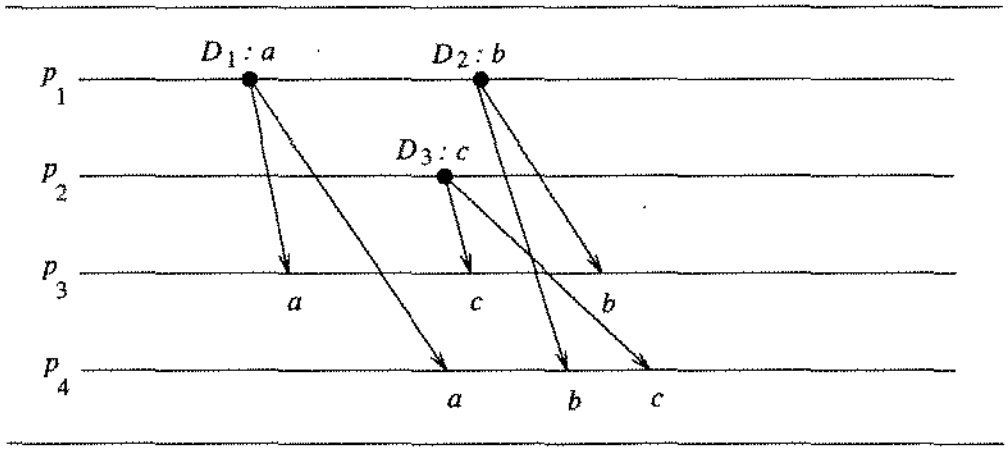


Figura 2.5: Ordenação FIFO.

2.3.3 Ordenação Total

Difusões são normalmente usadas para atualizar informações que são replicadas em diferentes nós de uma rede. Ordenação FIFO pode não ser suficiente se diferentes processos difundem mensagens de atualização independentemente. Nesta situação, duas mensagens de atualizações podem ser entregues em diferentes ordens por diferentes protocolos, criando inconsistências. Isto pode ser evitado usando uma ordenação mais rígida que garantirá que todas as mensagens são entregues na mesma ordem em todos os nós, mesmo se elas são enviadas independentemente por diferentes processos. Tal ordenação é chamada de *ordenação total*. A Figura 2.6 ilustra o comportamento de tal ordenação. Ela mostra duas difusões de mensagens independentes por p_1 e p_2. Ambas as mensagens são recebidas na mesma ordem por p_3 e p_4 (primeiro b e depois a neste exemplo).

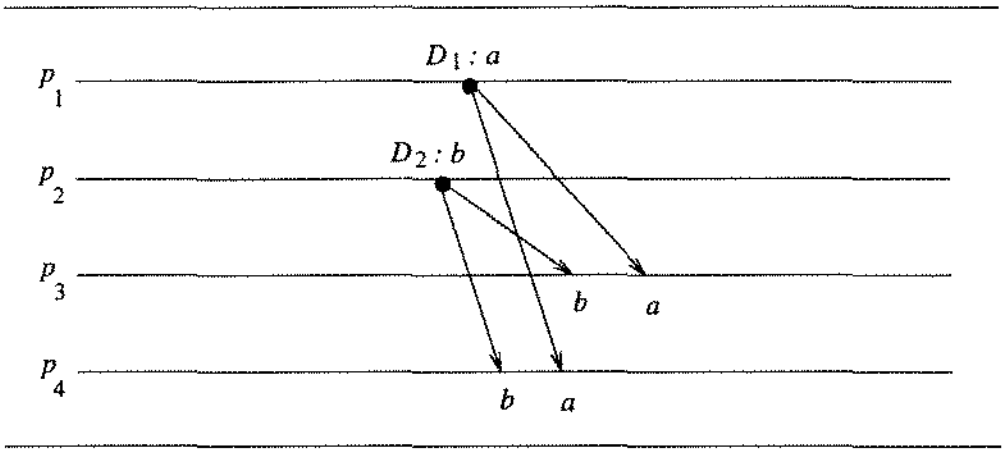


Figura 2.6: Ordenação Total.

Existem várias técnicas bem conhecidas para implementar ordenação total em um sistema assíncrono. Uma destas técnicas foi desenvolvida pelo protocolo de Chang e Maxemchuk [CM84], descrito no Capítulo 3, no qual cada mensagem é difundida em duas fases. Um processo que deseja difundir uma mensagem transmite esta mensagem por *broadcast* e um processo distinto, chamado de *nó ficha*, ao receber esta mensagem, difunde o número de ordem total da mensagem para todos os processos do grupo. Desta forma, todas as mensagens de difusão são entregues na ordem em que o *nó ficha* as "viu" e reconheceu.

Ordenação total torna o projeto de aplicações distribuídas tolerantes a falhas mais fácil, pois ela reduz as incertezas causadas por atrasos e falhas no envio das mensagens no sistema. Entretanto, este benefício tem um custo. A maioria dos protocolos que implementam ordenação total necessitam de duas ou mais fases de comunicação antes que uma mensagem possa ser entregue. Não é difícil provar que em um sistema assíncrono (isto é, um sistema com atrasos ilimitados no envio de mensagens), qualquer protocolo que garante difusão total requer algumas mensagens que tomam pelo menos duas fases (saltos[1]) antes que elas possam ser entregues.

[1] *hops* do inglês, significando estágios de transmissão.

Considere, por exemplo, um sistema com dois processos, p_1 e p_2. O processo p_1 difunde uma mensagem a; neste mesmo instante p_2 difunde b. Ambas as mensagens são endereçadas para ambos os processos. Nós afirmamos que ou a mensagem a necessita de pelo menos dois saltos (para p_2 e de volta para p_1) antes que ela possa ser entregue em p_1, ou a mensagem b necessita de dois saltos. Assuma que o protocolo entrega a para p_1 em um salto. Isto significa que p_1 envia a para p_2, mas ele entrega a mensagem localmente sem esperar por uma resposta de p_2 (veja Figura 2.7). No momento desta entrega local, p_1 pode não ainda saber que p_2 já enviou uma mensagem. Se a mensagem b de p_2 para p_1 é atrasada o suficiente, o protocolo entregará a antes de b em p_1. Similarmente, é possível que em p_2, b seja entregue antes de a. Mas isto violaria a ordenação total.

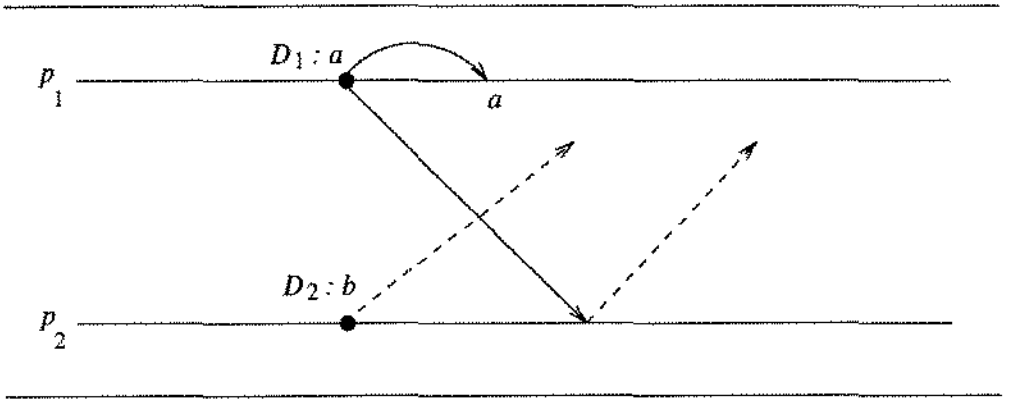


Figura 2.7: Inconsistência na ordem total de entrega.

A situação é diferente se existe um limite superior no atraso das mensagens. Em tais sistemas é possível manter relógios lógicos sincronizados. Neste caso, ordenação total pode ser conseguida por um método baseado em *timestamps*[2]. O remetente de uma difusão adiciona à mensagem um número seqüencial (*timestamp*) o qual descreve o valor do relógio lógico local quando a mensagem é enviada. As mensagens recebidas em um nó destino são entregues à aplicação na ordem deste timestamp; entretanto, antes de uma mensagem ser entregue, o processo tem que esperar até que ele esteja certo que nenhuma outra mensagem com um timestamp menor chegará. A quantidade de tempo a esperar depende do maior atraso para o envio de uma mensagem e de quão os relógios lógicos estão sincronizados. A desvantagem desta abordagem é que a entrega de toda mensagem é atrasada pelo maior atraso para envio de uma mensagem, o qual é normalmente muito maior que a média de atraso em um protocolo de duas ou mais fases.

2.3.4 Ordenação Causal

Em razão do custo inerente dos protocolos de ordenação total é natural procurar por protocolos que forneçam uma ordenação mais forte que ordenação FIFO, entretanto, menos cara que ordenação total. Os protocolos com *ordenação causal* apresentam tal propriedade. Eles são

[2]O termo *timestamp* será utilizado nesta dissertação como sinônimo para *marcador de tempo*.

baseados na idéia de *causalidade potencial* introduzida por Lamport em [Lam78].

O fluxo de informações durante a execução de um sistema distribuído pode ser usado para definir uma ordem parcial nos eventos que ocorrem no sistema. Tais eventos são *envio de uma mensagem*, *recebimento de uma mensagem*, ou um *evento local* que apenas afeta um único processo. A Figura 2.8 ilustra isto. Os eventos $\{e_1, e_4, e_{11}, e_{14}\}$ são eventos de envio e $\{e_2, e_5, e_7, e_8, e_{12}, e_{13}, e_{15}\}$ são eventos de recebimento. De acordo com a definição de Lamport, todos os eventos que estão conectados por um caminho neste diagrama são *relacionados causalmente em potencial*. Tal caminho deve seguir as linhas horizontais (da esquerda para a direita) ou as setas representando o envio de mensagens. Por exemplo, e_{10} é relacionado causalmente com e_1, pois há um caminho de e_1 até e_{10} através de e_2, e_4, e e_8 (linha pontilhada na figura). Esta dependência é denotada pelo símbolo "$\rightarrow$" (leia-se *precede*), e a definição formal é a seguinte:

1. Se existe o processo p, tal que o evento x aconteceu antes do evento y em p, então $x \rightarrow y$, ou x *precede* y.

2. Para qualquer mensagem m, $envia(m) \rightarrow recebe(m)$, onde $envia(m)$ é o evento de enviar a mensagem, e $recebe(m)$ é o evento de recebê-la.

3. Se x, y e z são eventos tais que $x \rightarrow y$ e $y \rightarrow z$, então $x \rightarrow z$.

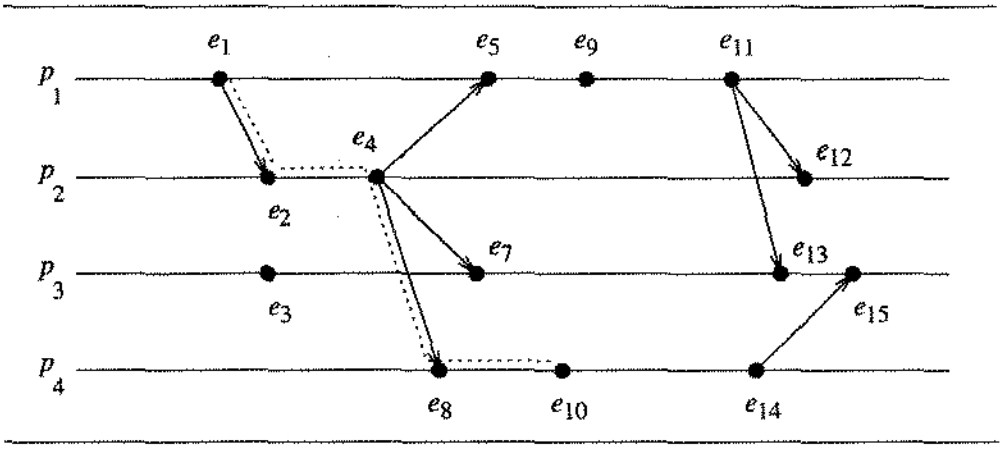


Figura 2.8: Causalidade Potencial.

Neste caso, temos que $e_1 \rightarrow e_{10}$. Eventos que não sejam conectados por tal caminho (linha pontilhada na Figura 2.8) são chamados de *concorrentes*. Isto é denotado pelo símbolo "//". Por exemplo, $e_5 // e_{14}$. A relação "$\rightarrow$" é chamada de *causalidade potencial* ou *relacionamento de fluxo de informação*. O nome "causalidade potencial" tem o seguinte significado. Na Física, o Princípio da Causalidade diz que uma causa tem que preceder seus efeitos. Similarmente, um evento a qualquer em um sistema distribuído pode afetar um evento b em algum outro processo apenas se há um fluxo de informação de a até b, isto é, se $a \rightarrow b$.

As propriedades de ordenação para um protocolo de difusão causal são definidas em termos deste relacionamento no fluxo de informações das mensagens. Ordenação causal garante que

todo processo recebe mensagens em uma ordem que é consistente com "→". Isto é, sempre que dois eventos de difusão causal são relacionados por "→", o protocolo assegura que os dois eventos são recebidos na mesma ordem em qualquer destino, definida pela relação "precede". Por exemplo, na Figura 2.9, as difusões D_1 e D_3 estão relacionadas causalmente ($D_1 \rightarrow D_3$, representada pela linha pontilhada na Figura 2.9). Conseqüentemente, a mensagem a é recebida antes de c tanto em p_3 quanto em p_4, como mostrado neste exemplo. Desta forma, a ordenação causal também respeita a ordenação FIFO. Por exemplo, na Figura 2.9, $D_1 \rightarrow D_4$; assim a mensagem a é recebida antes de d em qualquer destino.

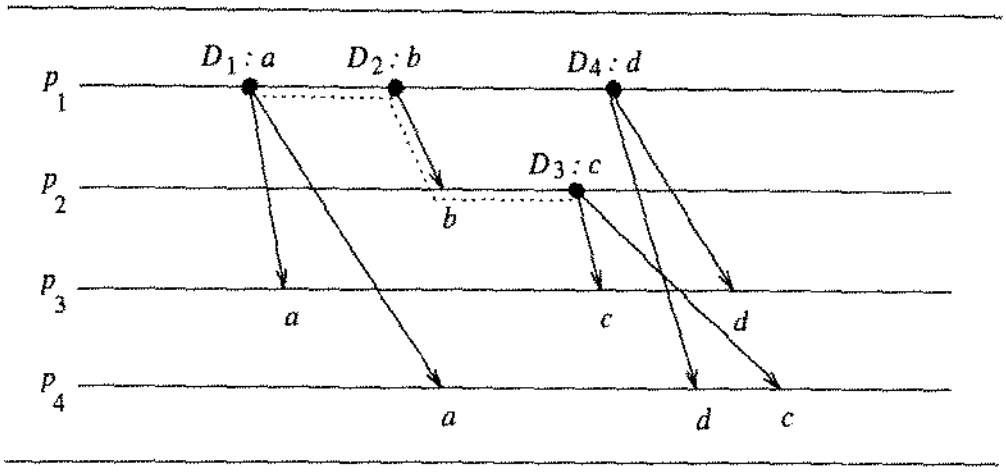


Figura 2.9: Ordenação Causal.

Existem várias formas de implementar ordenação causal através do uso de números seqüenciais (*timestamps*) nas mensagens enviadas. Um processo que deseja difundir uma mensagem adiciona alguma informação de dependência à mensagem antes de enviá-la aos seus destinos. A informação que é adicionada a uma mensagem difundida consiste de uma lista de outras mensagens, previamente recebidas, que precedem m sob a relação "→". Esta forma de protocolo com ordenação causal é descrita em detalhes por Birman e Joseph em [BJ87], e será vista no Capítulo 4. Em um sistema em que não ocorrem falhas, é suficiente transmitir apenas os identificadores das mensagens, ao invés de reconhecer todas as mensagens já recebidas nas mensagens seguintes [LLPS89]. Usando esta técnica de reconhecimento, ordenação causal pode ser conseguida sem a necessidade de múltiplas fases de trocas de mensagens.

2.4 Difusões Confiáveis

Os protocolos de ordenação descritos na seção anterior funcionam corretamente apenas se não ocorrem falhas no sistema. Considere por exemplo a difusão não-ordenada. Se o remetente pára no meio do protocolo, a mensagem poderá alcançar apenas um subconjunto dos processos destinos. A situação é ainda pior num protocolo com ordenação total com duas ou mais fases. A falha de um simples processo destino pode causar o bloqueio do protocolo, evitando que todas as outras difusões sejam recebidas.

Os protocolos de *difusão confiável* evitam estes comportamentos indesejáveis. Uma difusão confiável garante que toda mensagem enviada, ocasionalmente, será recebida por todos os processos destinos operacionais, apesar da ocorrência de falhas. Nós temos que especificar um pouco mais esta garantia. Sob certos tipos de falhas, nenhum protocolo pode garantir a entrega de uma difusão para todos os destinos operacionais. Por exemplo, o remetente pode falhar antes de enviar qualquer mensagem. Ainda, mesmo se o remetente conseguiu se comunicar com algum processo antes de falhar, este outro processo poderia também falhar antes de falar com qualquer outro processo. Em geral, um conjunto de falhas em um estágio inicial de um protocolo de difusão pode liquidar com qualquer conhecimento sobre a mensagem a ser enviada. O que nós queremos dizer com entrega confiável de mensagem é que uma mensagem é entregue a todos os destinos operacionais a menos que o remetente falhe antes que o protocolo tenha terminado. Além disso, no caso do remetente falhar em algum instante durante o protocolo, a entrega da mensagem pode ser *todos-ou-nenhum*. Mais precisamente:

> Se o processo p envia uma mensagem m para um conjunto D de nós destinos, então o sistema em um determinado momento alcançará um dos dois estados seguintes:
>
> 1. Para todo $q \in D$: q recebeu m ou q falhou.
> 2. O processo p falhou, e para todo $q \in D$: q falhou ou q nunca receberá m.

Esta propriedade também é chamada de *entrega atômica de mensagem*. Para fornecer esta propriedade, alguns protocolos analisados neste trabalho utilizam o conceito de *sincronismo virtual* que permite ao protocolo lidar com falhas dos processos e com particionamentos de rede.

2.5 Sincronismo Virtual Estendido

Sincronismo virtual estendido, introduzido em [LEMA94b], amplia o modelo inicial de sincronismo virtual do Sistema ISIS [SS91, Bir93]. Sincronismo virtual foi projetado no ISIS para suportar falhas que respeitam o modelo de falha de parada de processos. Em adição, sincronismo virtual estendido suporta falhas de parada e recuperação de processos, além de particionamento e junção da rede.

O significado do sincronismo virtual estendido é que, durante o particionamento e junção de rede e durante paradas e recuperações de processos, ele mantém um relacionamento consistente entre a entrega de mensagens e a entrega de notificações de mudanças na configuração para todos os processos no sistema.

Cada processo participando do grupo executa uma camada de comunicação em grupo (CG) tal como visto na Figura 2.3. Cada CG executa um protocolo de difusão confiável e um protocolo de gerenciamento do grupo. A comunicação com o meio físico de rede é gerenciado pela CG. O protocolo de gerenciamento do grupo determina quem são os processos membros da configuração atual do sistema. Estes membros, juntamente com um identificador único, são chamados de uma *configuração*. Uma configuração que é instalada por um processo, representa a visão deste

processo da conectividade no sistema. Cada processo é informado das mudanças na configuração através da entrega de mensagens de notificação de alterações do grupo.

Como discutido na seção 2.1, existe uma distinção entre a *recepção* de uma mensagem pela *camada CG*[3] sobre o meio de comunicação, a qual pode estar fora de ordem, e a *entrega* desta mensagem pela CG, que pode ser atrasada até que as mensagens anteriores na ordem definida sejam entregues. As mensagens podem ser entregues em ordem **concordada** e em ordem **segura**. *Entrega concordada* garante uma ordem total de mensagens entregues dentro de cada segmento de rede, permitindo que a mensagem seja entregue assim que todas as suas predecessoras na ordem total tenham sido entregues. *Entrega segura* requer em adição a isto, que se uma mensagem é entregue pela CG para qualquer processo em uma configuração, esta mensagem foi recebida e será entregue por cada processo nesta configuração a menos que este processo falhe e pare.

Para conseguir entrega segura na presença de particionamento e junção de rede, além de parada e recuperação de processos, sincronismo virtual estendido apresenta dois tipos de configuração. Em uma *configuração regular* as novas mensagens são enviadas e entregues. Em uma *configuração transacional* nenhuma mensagem nova pode ser enviada, mas as mensagens pertencentes a configuração regular anterior são entregues. As mensagens novas não satisfazem os requisitos de entrega segura na configuração regular, e assim, não podem ser entregues lá. Uma configuração transacional consiste dos membros da próxima configuração regular, os quais faziam parte da configuração anterior.

Uma configuração regular pode ser imediatamente seguida de várias configurações transacionais (uma para cada segmento da rede particionada) e podem ser imediatamente precedidas pelas várias configurações transacionais (quando os vários segmentos da rede se unem novamente). Uma configuração transacional, por outro lado, é imediatamente seguida e precedida por uma configuração regular única (em razão de que ela consiste somente dos membros da próxima configuração regular vindos diretamente da configuração anterior).

Para um processo p que é membro de uma configuração regular c, definimos $trans_p(c)$ como a configuração transacional que segue c em p, se tal configuração existe. Para um processo p que é membro de uma configuração transacional c, $trans_p(c) = c$. Para um processo p que é um membro de uma configuração transacional c, definimos $reg(c)$ como sendo a configuração regular que imediatamente precede c. Para um processo p que é um membro de uma configuração regular c, $reg(c) = c$. Definimos $com_p(c)$ como sendo ou uma das configurações, $reg(c)$ ou $trans_p(c)$. Note que se tanto b quanto q são membros de uma configuração regular c, $trans_p(c)$ não é necessariamente igual a $trans_q(c)$ e, assim, $com_p(c)$ não é necessariamente igual a $com_q(c)$.

Sincronismo virtual estendido é definido em termos de quatro tipos de eventos:

- $entrega\_conf_p(c)$: a CG entrega ao processo p uma mensagem de mudança na configuração iniciando a configuração c, onde p é um membro de c.
- $envia_p(m, c)$: a CG envia a mensagem m gerada por p enquanto p é um membro da configuração c.

[3]Camada de rede responsável por implementar a comunicação em grupo.

- $entrega_p(m, c)$: a CG entrega a mensagem m para p enquanto p é um membro da configuração c.
- $falha_p(c)$: o processo p pára enquanto p é um membro da configuração c.

O evento $falha_p(c)$ é a falha real do processo p na configuração c e é diferente de um evento $entrega\_conf_q(c')$ que remove p da configuração c no processo q. Depois de um evento $falha_p(c)$, o processo p pode permanecer parado para sempre, ou pode se recuperar com um $entrega\_conf_p(c'')$ onde a configuração c'' é $\{p\}$.

A relação precede, '$\rightarrow$', define uma ordem parcial em todos os eventos no sistema. A função *ord*, de eventos para números naturais, define uma ordem total lógica destes eventos. A função *ord* não é um-para-um, porque alguns eventos em processos diferentes ocorrem no mesmo tempo lógico. A semântica do sincronismo virtual estendido a seguir define a relação '$\rightarrow$' e a função *ord*.

2.5.1 Semântica do Sincronismo Virtual Estendido

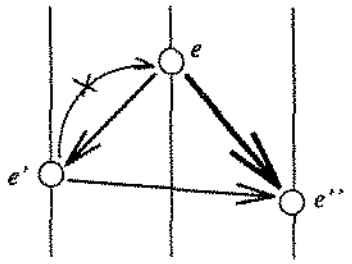
A semântica do sincronismo virtual estendido consiste das Especificações 1 a 7 abaixo. Nas figuras usadas como ilustração, as linhas verticais correspondem aos processos, um círculo aberto representa um evento que *assume-se* existir, um triângulo representa um evento que é *certo* existir, uma linha fina sem uma seta representa uma relação de precedência que é *validada* em razão de alguma outra especificação, uma linha média com uma seta representa um a relação de precedência que é *assumida* entre dois eventos, uma linha grossa com uma seta representa uma relação de precedência que é *validada* entre dois eventos, e um cruzamento de linhas sobre um evento ou relação indica que este evento/relação não ocorre. Em todas as figuras, o tempo progride no sentido para baixo das linhas dos processos.

Entrega Básica

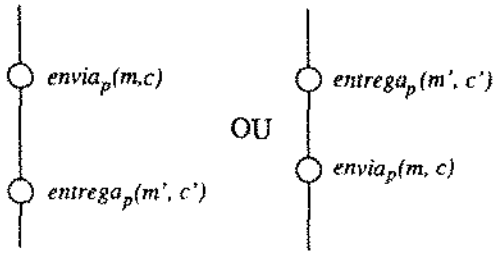
A Especificação 1.1 requer que '$\rightarrow$' seja uma relação irreflexiva, anti-simétrica e transitiva. A Especificação 1.2 requer que os eventos dentro de um único processo sejam totalmente ordenadas pela relação '$\rightarrow$'. A Especificação 1.3 requer que o envio de uma mensagem preceda a sua entrega, e esta entrega ocorra na configuração na qual a mensagem foi enviada ou em uma configuração transacional imediatamente seguinte. A Especificação 1.4 afirma que uma determinada mensagem não é enviada mais de uma vez e não é entregue em duas configurações diferentes para o mesmo processo.

1.1 Para qualquer evento e, e não precede ele mesmo.
Se existe os eventos e e e', tais que $e \rightarrow e'$, não pode haver o caso que $e' \rightarrow e$.
Se existe os eventos e, e' e e'' tais que $e \rightarrow e'$ e $e' \rightarrow e''$, então $e \rightarrow e''$.

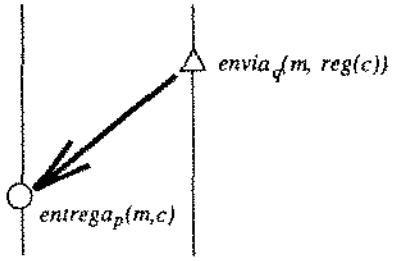
1.2 Se existe um evento e que é ou $entrega\_conf_p(c)$ ou $envia_p(m, c)$ ou $entrega_p(m, c)$ ou $falha_p(c)$, e um evento e' que é ou $entrega\_conf_p(c')$ ou $envia_p(m', c')$ ou $entrega_p(m', c')$ ou $falha_p(c')$, então $e \rightarrow e'$ ou $e' \rightarrow e$.



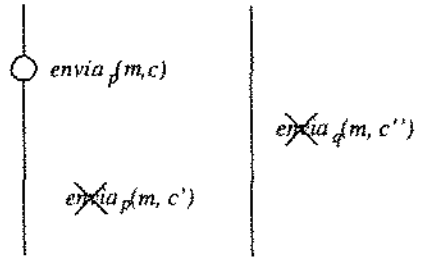
Especificação 1.1



Especificação 1.2



Especificação 1.3



Especificação 1.4

Figura 2.10: Especificações da Entrega Básica.

1.3 Se existe $entrega_p(m, c)$, então existe $envia_q(m, reg(c))$ tal que $envia_q(m, reg(c)) \rightarrow entrega_p(m, c)$.

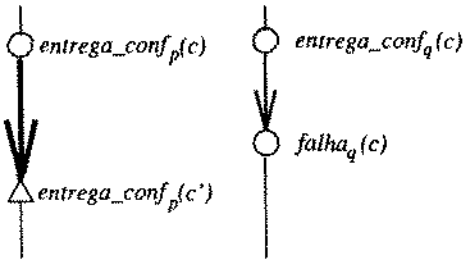
1.4 Se existe $envia_p(m, c)$, então $c = reg(c)$ e não existe $envia_p(m, c')$ onde $c \neq c'$, nem $envia_q(m, c'')$ onde $p \neq q$.

Além disso, se existe $entrega_p(m, c)$, então não existe $entrega_p(m, c')$ onde $c \neq c'$.

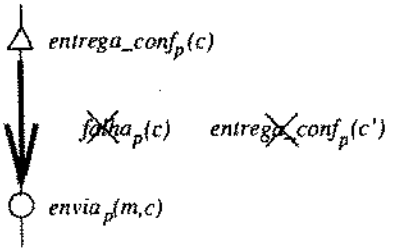
Entrega de Mensagens de Mudanças na Configuração

A Especificação 2.1 requer que se um processo falha ou a rede se particiona, então a CG detecta isto e entrega uma nova mensagem de mudança na configuração para os outros processos pertencentes à configuração antiga. A Especificação 2.2 declara que em qualquer momento um processo é um membro de uma única configuração cujos eventos são delimitados pelo(s) evento(s) de mudança para esta configuração. As Especificações 2.3 e 2.4 afirmam que um evento que precede a entrega de uma mudança de configuração para um processo deve também preceder a entrega desta mudança para os outros processos.

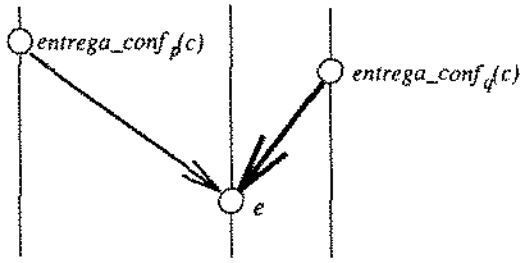
2.1 Se existe $entrega\_conf_p(c)$ e não existe $falha_p(c)$ e nem existe $entrega\_conf_p(c')$ tal que $entrega\_conf_p(c) \rightarrow entrega\_conf_p(c')$, e se q é um membro de c, então existe $entrega\_conf_q(c)$, e não existe $falha_q(c)$ e nem $entrega\_conf_q(c'')$ tal que $entrega\_conf_p(c) \rightarrow entrega\_conf_q(c'')$.



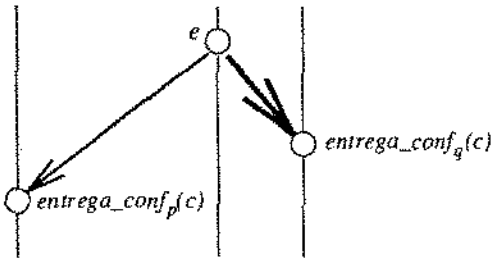
Especificação 2.1



Especificação 2.2



Especificação 2.3



Especificação 2.4

Figura 2.11: Especificações de Mudanças na Configuração.

2.2 Se existe um evento e que é ou $envia_p(m,c)$, $entrega_p(m,c)$ ou $falha_p(c)$, então existe $entrega\_conf_p(c)$ tal que $entrega\_conf_p(c) \rightarrow e$, e não existe um evento e' tal que e' é $falha_p(c)$ ou é $entrega\_conf_p(c')$ e $entrega\_conf_p(c) \rightarrow e' \rightarrow e$.

2.3 Se existe $entrega\_conf_p(c)$, $entrega\_conf_q(c)$ e e tal que $entrega\_conf_p(c) \rightarrow e$, então $entrega\_conf_q(c) \rightarrow e$.

2.4 Se existe $entrega\_conf_p(c)$, $entrega\_conf_q(c)$ e e tal que $e \rightarrow entrega\_conf_p(c)$, então $e \rightarrow entrega\_conf_q(c)$.

Auto-Entrega

A Especificação 3 requer que cada mensagem que é gerada por um processo é entregue para este processo, contanto que ele não falhe. Além disso, a mensagem é entregue na mesma configuração em que ela foi enviada, ou na configuração transacional que segue.

3 Se existe $envia_p(m,c)$ e $entrega\_conf_p(c')$ onde $c' \neq trans_p(c)$, tal que $envia_p(m,c) \rightarrow entrega\_conf_p(c')$, e não existe $falha_p(com_p(c))$, **então** existe $entrega_p(m, com_p(c))$.

Atomicidade de Falhas

A Especificação 4 requer que se quaisquer dois processos progridem juntos de uma configuração para a próxima, a CG entrega o mesmo conjunto de mensagens para ambos os processos nesta configuração.

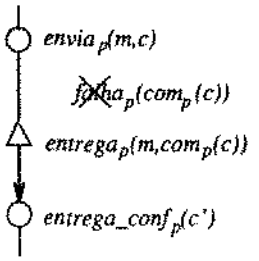


Figura 2.12: Especificação da Auto-Entrega.

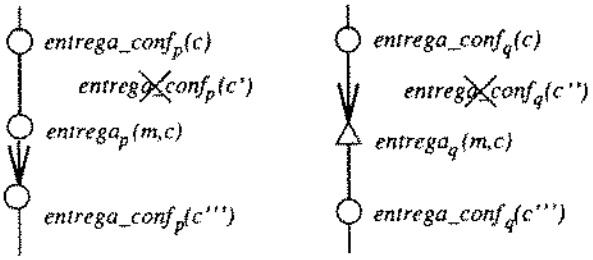


Figura 2.13: Especificação da Atomicidade de Falhas.

4 Se existe $entrega\_conf_p(c)$, $entrega\_conf_p(c''')$, $entrega\_conf_q(c)$, $entrega\_conf_q(c''')$ e $entrega_p(m,c)$, tais que $entrega\_conf_p(c) \rightarrow entrega\_conf_p(c''')$, e não existe $entrega\_conf_p(c')$ tal que $entrega\_conf_p(c) \rightarrow entrega\_conf_p(c') \rightarrow entrega\_conf_p(c''')$ e não existe $entrega\_conf_q(c'')$ tal que $entrega\_conf_q(c) \rightarrow entrega\_conf_q(c'') \rightarrow entrega\_conf_q(c''')$ **então** existe $entrega_q(m,c)$.

Entrega Causal

Modelamos causalidade de forma que esta é local para uma única configuração e é finalizada por uma mensagem de mudança de configuração. Formulações mais simples [Lam78, Bir93] não são apropriadas quando uma rede pode se particionar e se re-juntar ou quando um processo pode falhar e se recuperar em seguida. O relacionamento causal entre mensagens é expresso na Especificação 5 como uma relação de precedência entre o envio de duas mensagens na mesma configuração. Esta relação de precedência está contida no fechamento transitivo das relações 'precede' estabelecidas pelas Especificações 1.1 a 1.3.

A Especificação 5 requer que se uma mensagem é enviada antes de uma outra na mesma configuração e se a CG entrega a segunda destas mensagens, então ela também entrega a primeira.

5 Se existe $envia_p(m,c)$, $envia_q(m',c)$ e $entrega_r(m', com_r(c))$ tais que $envia_p(m,c) \rightarrow envia_q(m',c)$, então existe $entrega_r(m, com_r(c))$ tal que $entrega_r(m, com_r(c)) \rightarrow entrega_r(m', com_r(c))$.

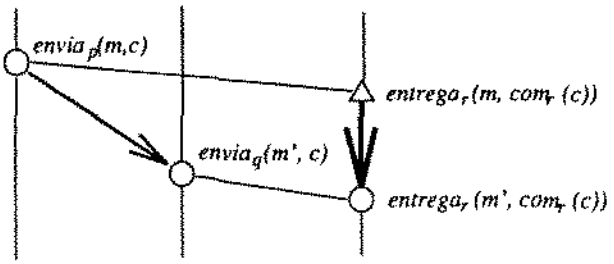
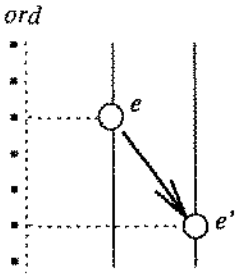


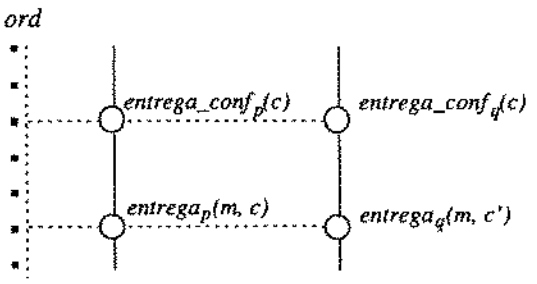
Figura 2.14: Especificação da Entrega Causal.

Entrega Concordada

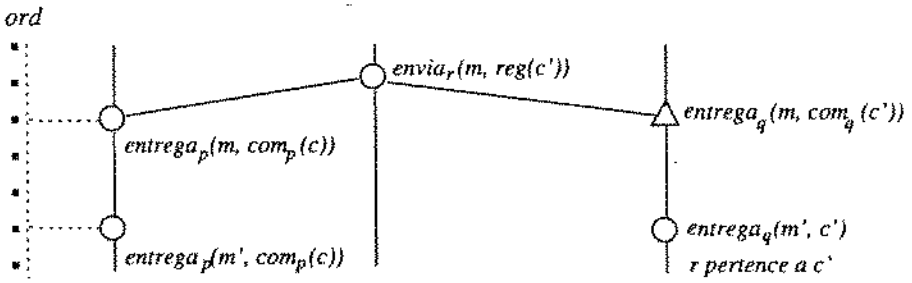
As seguintes especificações contém a definição da função *ord*. A Especificação 6.1 requer que a ordem total seja consistente com a ordem parcial. A Especificação 6.2 afirma que a CG entrega mensagens de mudança de configuração para a mesma configuração, no mesmo tempo lógico para cada processo. As mensagens são também entregues no mesmo tempo lógico para cada processo, seja qual for a configuração em que elas são entregues. A Especificação 6.3 requer que a CG entregue mensagens em ordem para todos os processos exceto que, na configuração transacional não há obrigação de entregar mensagens geradas por processos que não são membros desta configuração transacional.



Especificação 6.1



Especificação 6.2



Especificação 6.3

Figura 2.15: Especificações da Entrega em Ordem Total.

6.1 Se existe os eventos e e e' tais que $e \rightarrow e'$, então $ord(e) < ord(e')$.

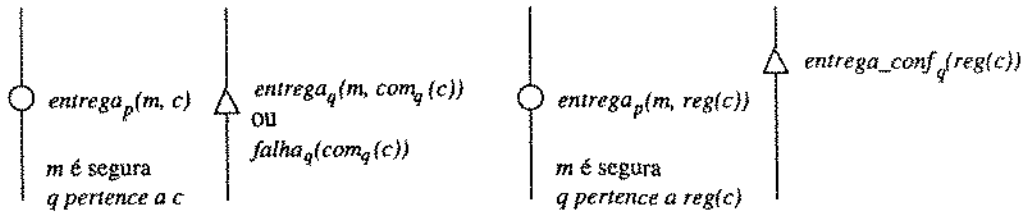
6.2 Se existe os eventos e e e' que são ou $entrega\_conf_p(c)$ e $entrega\_conf_q(c)$ ou $entrega_p(m, c)$ e $entrega_p(m, c')$, então $ord(e) = ord(e')$.

6.3 Se existe $entrega_p(m, com_p(c))$, $entrega_p(m', com_p(c))$, $entrega_q(m', c')$ e $envia_r(m, reg(c'))$ tal que $ord(entrega_p(m, com_p(c))) < ord(entrega_p(m', com_p(c)))$ e r é um membro de c', então existe $entrega_q(m, com_q(c'))$.

Note que o relacionamento entre c e c' na Especificação 6 pode apenas ser um dos seguintes: ou eles são a mesma configuração regular ou transacional, ou eles são configurações transacionais diferentes para a mesma configuração regular, ou um é uma configuração regular e o outro é uma configuração transacional que a segue.

Entrega Segura

A Especificação 7.1 requer que, se a CG entrega uma mensagem para um processo que está em uma configuração, então a CG entrega a mensagem para todos os processos nesta configuração a menos que o processo falhe, isto é, mesmo se a rede se particiona neste ponto, a mensagem ainda assim é entregue. A Especificação 7.2 afirma que, se a CG entrega uma mensagem do tipo *segura* para qualquer um dos processos em um nova configuração regular, então a CG entrega a mensagem de mudança para esta configuração para todos os membros desta configuração.



Especificação 7.1 Especificação 7.2

Figura 2.16: Especificações da Entrega Segura.

7.1 Se existe $entrega_p(m, c)$ para uma mensagem segura m, então para todo processo q em c existe ou $entrega_q(m, com_q(c))$ ou $falha_q(com_q(c))$.

7.2 Se existe $entrega_p(m, reg(c))$ para uma mensagem segura m, então para todo processo q em $reg(c)$ existe $entrega\_conf_q(reg(c))$.

Exemplo

O seguinte exemplo, mostrado na Figura 2.18, foi tirado de [Ami95]. Considere que uma configuração regular contendo p, q e r se particiona e p torna-se isolado enquanto q e r se unem em uma nova configuração regular com s e t. Quando ainda em $\{p, q, r\}$, cinco mensagens seguras foram enviadas na seguinte ordem: m_1 foi enviada por p, m_2 por q, m_3 por p, m_4 por r e m_5 foi

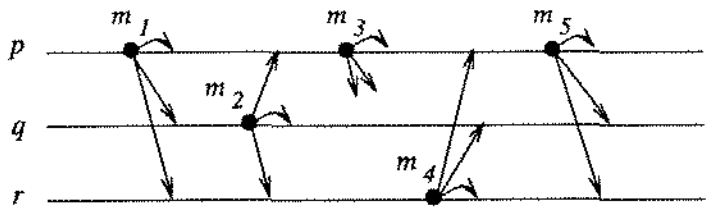


Figura 2.17: Um exemplo de envio e recebimento de mensagens.

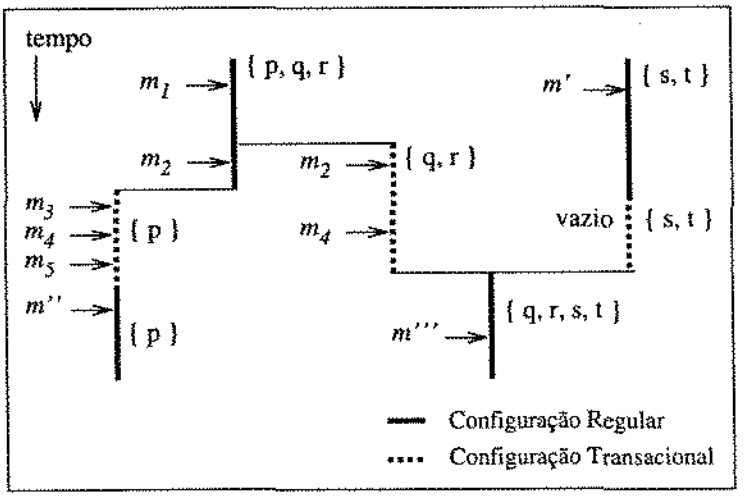


Figura 2.18: Mudanças na configuração e entrega de mensagens.

enviada por p, como visto na figura 2.17. p, q e r podem deduzir que m_1 foi recebida por todos eles.

Em p, todas as cinco mensagens foram recebidas. p pode deduzir que m_2 também foi recebida por q e r. Portanto, m_1 e m_2 satisfazem os requisitos de entrega segura e são entregues em p na configuração regular $\{p, q, r\}$. Entretanto, p não pode dizer se m_3, m_4 e m_5 foram recebidas por todos os membros de $\{p, q, r\}$. Assim, uma configuração regular $\{p\}$ é entregue em p seguida de m_3, m_4, m_5 e pela próxima configuração regular $\{p\}$.

Em q e r, apenas quatro mensagens foram recebidas: m_1, m_2, m_4 e m_5. Já que q e r sabem que p é requerido para entregar m_1, m_1 satisfaz os requisitos de entrega segura e é entregue em q e r na configuração regular $\{p, q, r\}$. Entretanto, q e r não podem deduzir que m_2 foi recebida em p. Assim, uma configuração transacional $\{q, r\}$ é entregue tanto em q quanto em r seguida por m_2.

A mensagem m_3 que foi enviada por p foi omitida por q e r e não foi recuperada antes da mudança de configuração ocorrer. Daí, m_4 é entregue em q e r imediatamente depois de m_2. Embora m_5 foi recebida por q e r, eles **não podem** entregá-la, pois m_5 pode vir causalmente depois de m_3 (o que é verdade neste exemplo) e eles não podem satisfazer os requisitos de entrega causal. Em seguida, a próxima configuração regular $\{q, r, s, t\}$ é entregue em q e r de forma que eles se unem com s e t.

Em s e t, todas as mensagens que foram enviadas antes da mudança de configuração que os uniu com q e r podem satisfazer os requisitos de entrega segura. Assim, todas as mensagens que foram enviadas na configuração regular $\{s, t\}$, tais como m', são entregues na configuração regular $\{s, t\}$. Em seguida, uma configuração transacional $\{s, t\}$ é entregue, e a próxima configuração regular $\{q, r, s, t\}$ também é entregue. Note que a configuração transacional é sempre vazia quando uma junção ocorre mas nenhum processo da configuração anterior se particiona ou falha.

Note, também, que ao entregar a configuração transacional, q e r cumprem com os requisitos de entrega concordada embora eles não possam entregar m_3. Esta é a maior diferença entre o sincronismo virtual padrão e o estendido. Usando o sincronismo virtual, pelo menos um dos dois seguimentos $\{q, r\}$ e $\{p\}$ teriam que bloquear e perder sua memória por causa da inconsistência potencial que ocorre quando p entrega m_3 em $\{p, q, r\}$, enquanto que q e r não o fazem. Sincronismo virtual estendido permite que ambos os seguimentos continuem, enquanto lhes fornecem informações úteis sobre o estado das mensagens em ambos os seguimentos.

2.5.2 Discussão

A Especificação de Entrega Básica 1.2, quando restringida para uma simples configuração, expressa a causalidade dos eventos dentro de um único processo.

Enquanto a Especificação 2.3 e 2.4 requerem mensagens de mudança de configuração para definir um "corte" consistente na ordem dos eventos em todos os processos, estes processos não são obrigados a recuperarem mensagens enviadas nas configurações que eles não pertenciam. A Especificação 5 limita os requisitos de entrega causal para a mesma configuração, eliminando a

necessidade de recuperar o histórico das configurações anteriores em outros processos a fim de satisfazer a causalidade.

Tradicionalmente, as definições de causalidade incluem, em adição à Especificação 5, uma especificação similar com $envia_p(m, c)$ substituído por $entrega_q(m, c)$. Note que esta nova especificação pode ser derivada das especificações existentes 5 e 1.3.

As Especificações 5 a 7 representam um aumento de nível dos serviços. Alguns sistemas podem operar sem os requisitos de ordem causal; outros necessitam destes requisitos e podem adicionar os requisitos de ordenação total e até mesmo os requisitos de entrega segura, de forma que isto atenda às necessidades da aplicação.

Nesta dissertação, apesar de alguns protocolos estudados abordarem mudanças no grupo através do sincronismo virtual estendido, os respectivos algoritmos não serão descritos, pois está fora do escopo principal deste trabalho.

2.6 Medidas de Eficiência dos Protocolos

Na seção anterior, foram consideradas características de tolerância a falhas dos protocolos de difusão confiável. Outro aspecto importante na avaliação desses protocolos é a sua eficiência. Tolerância a falhas e eficiência são objetivos conflitantes entre si em sistemas distribuídos de modo geral. O problema de se conseguir uma consistência global no sistema com comunicação em grupo não constitui exceção. A maioria dos autores baseia-se nos seguintes parâmetros para medida de eficiência dos protocolos de difusão:

1. **Número de mensagens por difusão**: Normalmente, em uma simples difusão de mensagem, o protocolo necessita de algumas mensagens adicionais para se manter a consistência global no sistema. Estas mensagens podem ser reconhecimentos do recebimento ou informações sobre a ordem da mensagem. Obviamente, estas mensagens adicionais têm um custo, e quanto menos forem geradas melhor será o desempenho do protocolo.

   Devemos considerar, também, que quase sempre o tamanho de uma mensagem de controle é menor que o de uma mensagem de difusão. Assim, ao determinarmos o número de mensagens geradas por difusão, devemos observar o que é controle e o que é difusão.

2. **Tempo médio necessário para se difundir uma mensagem**: Esta é uma medida um pouco cautelosa para se medir a eficiência de um protocolo. Isto porque, depende muito da topologia de rede a ser utilizada, da velocidade do canal de comunicação, do tamanho das mensagens, etc.. Muitos dos protocolos estudados aqui podem ser implementados em diferentes ambientes de redes, complicando ainda mais esta medida de eficiência.

   Tomaremos como tempo médio de execução do protocolo, o tempo decorrido entre o envio de uma difusão pela origem, até a entrega da mensagem por todos do grupo.

2.7 Classificação dos Protocolos

Nós criamos uma classificação para os protocolos baseando-se nas suas principais estruturas de controle e de dados para se alcançar a difusão confiável. Dentre os protocolos estudados nós observamos que existem protocolos baseados em:

- Anel e Ficha (descritos no próximo Capítulo);
- Vetor de Timestamps (descritos no Capítulo 4);
- Vetor de Timestamps com Sincronização de Fases (descritos no Capítulo 4);
- Grafo de Contexto (descritos no Capítulo 5);
- Grafo de Propagação (descritos no Capítulo 6); e
- Redundância de Hardware (descritos no Capítulo 6).

Capítulo 3

Protocolos baseados em Ficha/Anel

Como veremos a partir deste capítulo, os protocolos aqui discutidos são representações de diferentes abordagens de algoritmos para difusão. Os dois primeiros protocolos, descritos neste capítulo, se baseiam na passagem de uma *ficha* entre os processos membros do grupo, a qual é utilizada para designar o processo que manterá algum controle na distribuição das mensagens de difusão.

3.1 Protocolo de Chang e Maxemchuk

No modelo desenvolvido por Chang e Maxemchuk [CM84], as mensagens são difundidas através de *broadcast*. Um grupo de comunicação consiste de N processos; cada um dos processos pode transmitir e receber mensagens de difusão endereçadas para este grupo. É possível também a comunicação ponto-a-ponto entre estes processos. Uma mensagem recebida em um processo só é entregue à aplicação quando esta mensagem for recebida por todos os outros processos operacionais (ativos).

A filosofia principal do protocolo de Chang e Maxemchuk é fazer um sistema geral parecer como a combinação de outros dois sistemas simples. Um no qual existe um simples receptor e muitos transmissores, e um outro com um simples transmissor e muitos receptores. Esta idéia pode ser implementada através de um receptor primário, chamado de *nó ficha*, que se encarrega de receber a mensagem de difusão, estampá-la um número de seqüência (um *timestamp*) e difundir este timestamp para todos os outros processos como um reconhecimento da mensagem. Isto permite que todos os membros do grupo possam ordenar totalmente as mensagens de difusão que chegam.

Descrição do Protocolo

O sistema opera com um reconhecimento positivo entre os transmissores e o nó ficha, e com um reconhecimento negativo entre o nó ficha e os receptores, como pode ser visto na Figura 3.1. O timestamp que o nó ficha difunde funciona como um reconhecimento para o transmissor.

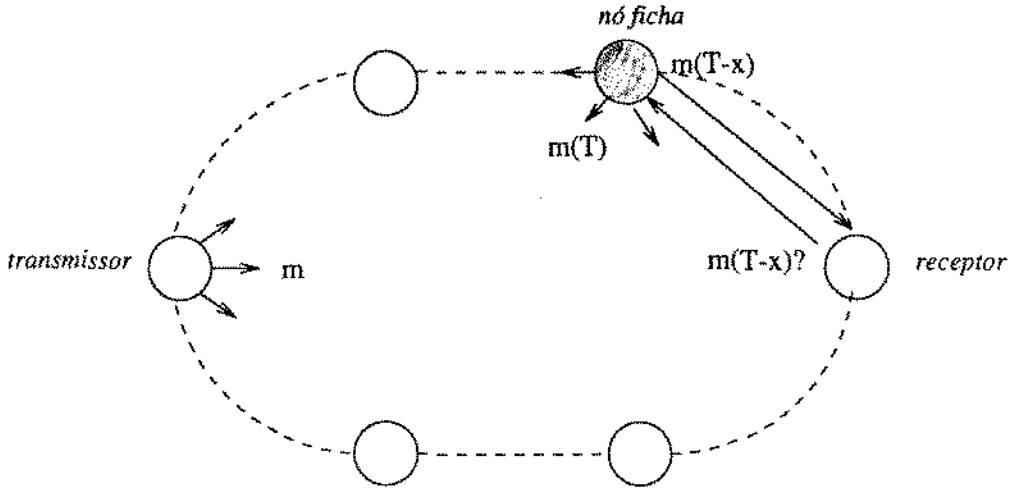


Figura 3.1: Difusões dos membros do grupo sendo reconhecidas pelo *nó ficha*.

Já os receptores usam este timestamp para detectarem as mensagens não recebidas; eles então requisitam ao nó ficha as mensagens que estão faltando, com seus respectivos timestamps.

Existem duas deficiências básicas com esta modelo de difusão: (1) Em um sistema com reconhecimento negativo, não há maneira de descobrir quando os receptores obtém as mensagens. (2) Quando o nó ficha falha, todo o controle das difusões pode ser perdido.

Para evitar estes problemas, o protocolo define três regras:

R1 A responsabilidade do nó ficha passará entre todos os processos operacionais através de um anel lógico previamente configurado, contendo todos os membros do grupo (Figura 3.1);

R2 Um receptor só poderá aceitar a ficha quando ele tiver todas as mensagens de timestamps, e quando já tiver recebido todas as mensagens difundidas; e

R3 Pelo menos L receptores adicionais aceitaram a ficha antes da entrega de uma mensagem. L é um parâmetro informado ao algoritmo e determina o grau de resistência a falhas do sistema.

Com estas regras, o protocolo garante que quando um processo aceita a ficha por uma segunda vez, todas as mensagens reconhecidas (estampadas) por ele da última vez que ele tinha a ficha, foram recebidas por todos os outros processos operacionais. E ainda, se L processos aceitaram a ficha depois que uma mensagem foi reconhecida, pelo menos $L + 1$ processos têm a mensagem, e pelo menos $L + 1$ processos teriam que falhar para a mensagem ser perdida.

O sistema alterna entre duas fases, uma fase normal e uma fase de reforma. Durante a fase normal, as mensagens são estampadas (reconhecidas) e a ficha é passada entre os processos. Quando uma falha é detectada ou um processo é recuperado, o sistema entra na fase de reforma. Durante esta fase, uma nova lista dos processos operacionais no anel, é gerada.

Estruturas de Dados

Cada processo p_i mantém as seguintes variáveis:

- tl_i: Número de versão da configuração atual do anel;
- $M_i[p_j]$: Número de seqüência da próxima mensagem de difusão que ele espera do processo p_j;
- nts_i: Próximo timestamp que ele espera receber.
- Qb_i: Fila de mensagens de difusão recebidas.
- Qc_i: Fila de mensagens de reconhecimento recebidas.

Durante a fase normal, um processo fica pertencendo a apenas uma configuração do anel tl_i, e apenas os processos desta mesma configuração se comunicam entre si. $M_i[p_j]$ é usado para diferenciar as novas mensagens do processo p_j das mensagens que já foram recebidas e reconhecidas. O propósito de nts_i é assegurar que o processo p_i obtenha todas as mensagens de reconhecimento na seqüência apropriada, ou seja, na ordem total determinada pelo algoritmo.

O Algoritmo

Há três fases no processo de difusão de uma mensagem: transmissão, reconhecimento e entrega. O pseudocódigo executado pelos processos em cada fase é visto no Algoritmo 3.1.

1. *Transmissão.* Um processo transmite continuamente uma mensagem de difusão até que ele receba um reconhecimento (*acknowledgement*, ou simplesmente, ACK) para esta mensagem (linha 2 do algoritmo). Se o nó ficha não transmite nenhuma mensagem durante R retransmissões sucessivas, o processo pode assumir que o nó ficha falhou. R representa um número arbitrário que também deve ser informado ao algoritmo.

   Cada mensagem de difusão é identificada por $m(p_i, n)$, onde p_i é o processo que a originou e $n = M_i[p_i]$ é o número da mensagem deste processo. Uma nova mensagem de difusão não pode ser transmitida até que o reconhecimento seja recebido.

2. *Reconhecimento.* O nó ficha reconhece mensagens de difusão. Quando o nó ficha p_t recebe uma difusão $m(p_i, n)$ para a qual $M_t[p_i] = n$ (linha 3), ele sabe que esta mensagem ainda não foi difundida. Ele então transmite uma mensagem de reconhecimento, $ACK(nts_t, m(p_i, n), p_k)$, onde p_k é a identificação do processo que será o próximo nó ficha. A fim de garantir que uma mensagem já reconhecida possa ser entregue, a ficha deve ser transmitida L vezes (regra R3). Assim, ocasionalmente um processo transfere a ficha quando não há nenhuma mensagem de difusão a ser enviada em um certo período de tempo T (linha 8). Para isto, utiliza-se um temporizador, e ao expirar o tempo T, o nó ficha envia uma mensagem de reconhecimento "nula": $ACK(nts_t, NULL, p_k)$.

Transmissão:

O processo p_i ao difundir uma mensagem m:

$n := M_i[p_i]$; 1

transmita $m(p_i, n)$ até R vezes, senão assuma que o nó ficha falhou; 2

Reconhecimento:

O nó ficha p_t ao receber uma mensagem $m(p_i, n)$:

se $M_t[p_i] = n$ então 3

$p_k :=$ *próximo_processo_do_anel*; 4

transmita $ACK(nts_t, m(p_i, n), p_k)$; 5

$M_t[p_i] := M_t[p_i] + 1$; 6

O nó ficha p_t ao passar um tempo T sem receber mensagens de difusão:

$p_k :=$ *próximo_processo_do_anel*; 7

transmita $ACK(nts_t, NULL, p_k)$; 8

O processo p_j ao receber $m(p_i, n)$:

se $m \notin Qb_j$ então insira $m(p_i, n)$ em Qb_j; 9

O processo p_j ao receber $ACK(ts, \{(m(p_i, n)) \vee (NULL)\}, p_k)$:

insira ACK em Qc_j; 10

para $(\forall ACK(ts, \{(m(p_i, n)) \vee (NULL)\}, p_k) \in Qc_j)$ faça 11

se $(ts = nts_j) \wedge ((m \in Qb_j) \vee (m = NULL))$ então 12

$nts_j := nts_j + 1$; 13

se $(m(p_i, n) \in Qb_j)$ então $M_j[p_i] := n + 1$; 14

senão

se $(m \notin Qb_j)$ então 15

transmita pedido de retransmissão por $m(p_i, n)$ até que a obtenha; 16

insira os ACK's que chegam neste período em Qc_j após o reconhecimento ts; 17

senão

se $(ts < nts_j)$ então $\{ACK\ já\ processado\}$ 18

descarte este reconhecimento; 19

senão

se $(ts > nts_j)$ então $\{ACK's\ perdidos\}$ 20

transmita pedidos de retransmissão dos ACK's entre

nts_j e $ts - 1$ até que os obtenha; 21

insira estes ACK's que chegam em Qc_j antes do reconhecimento ts; 22

Validação:

A cada transmissão do token:

para $(\forall m(p_i, n) \in Qb$, tal que, m não foi entregue, faça 23

$m(p_i, n).tpc := m(p_i, n).tpc + 1$; 24

se $(ACK(ts, m(p_i, n), p_k) \in Qc)$ então 25

se $(m(p_i, n).tpc = L + 1)$ então entregue m; 26

Algoritmo 3.1: Protocolo de Chang e Maxemchuk.

Todos os processos receptores armazenam as mensagens de difusão, $m(p_i, n)$, na fila Qb e processam os reconhecimentos, $ACK(ts, m, p_k)$, na ordem em que eles são recebidos e inseridos em Qc (linhas 10 a 22). No processo p_j, $ACK(ts, m, p_k)$ só é processado quando $nts_j = ts$ e $m = NULL$ ou $m = m(p_i, n)$ está em Qb_j (linha 12). Quando o reconhecimento ts é processado, nts_j é incrementado e se uma mensagem foi reconhecida ($m(p_i, n) \in Qb_j$), o número da próxima mensagem da origem p_i é atualizada para $M_j[p_i] = n + 1$. Se $ts < nts_j$, então este reconhecimento já foi processado. Se $ts > nts_j$, então mensagens de reconhecimento foram perdidas. Antes de processar o reconhecimento ts, os reconhecimentos faltantes são requisitados e processados. Se a mensagem de difusão sendo reconhecida, $m(p_i, n)$, não está em Qb_j, ela então deve ser obtida antes que seu reconhecimento seja processado. Pedidos de retransmissão são enviados até que a mensagem requisitada seja recebida, ou até que uma falha ocorra. Todas as difusões e reconhecimentos que chegam, enquanto uma mensagem requisitada é esperada, são armazenados em Qb_j e Qc_j, respectivamente, e processados posteriormente na devida ordem.

3. *Validação.* Assuma que cada mensagem de difusão contém um campo, *tpc* (*token passing count*), que conta o número de passagens da ficha após o seu reconhecimento. Depois que a mensagem recebe o timestamp e a ficha é transferida L vezes, é certo que $L + 1$ processos obtiveram a mensagem, e neste momento ela pode ser entregue (linha 26). Se no máximo L processos falharem, todas as mensagens entregues podem ser recuperadas. Isto caracteriza um sistema *L-resistente*.

Na transferência da ficha, o próximo processo do anel deve enviar alguma mensagem indicando que ele já tem todas as mensagens difundidas e conseqüentemente, ele aceita a ficha (Regra R2). Esta indicação pode ser feita quando o próximo nó ficha recebe uma difusão e transmite o reconhecimento para a mesma, ou através de uma mensagem de aceitação da ficha caso não tenha novas difusões por um certo período.

Fase de Reforma

O protocolo entra na fase de reforma quando uma falha ou uma recuperação é detectada. A lista de processos (*token list*) inicialmente contém todos os processos do grupo. A falha de algum destes processos pode corromper o mecanismo de passagem da ficha. Daí, a fase de reforma redefine a lista de processos. A nova lista consistirá apenas dos processos ativos, não-falhos. Quando uma nova lista é formada, o protocolo continua a sua operação normal.

Qualquer processo que detecta uma falha inicia a fase de reforma, e este é chamado de *iniciador*. Já que pode haver diferentes listas em instantes diferentes, um *número de versão* é associado com cada lista. Uma nova lista terá sempre um número maior que os da versão das listas antigas. Como vários processos podem, ao mesmo tempo, iniciar a fase de reforma e como falhas podem acontecer mesmo nesta fase, o protocolo tem que assegurar que:

1. Apenas uma lista de processos válida pode existir em qualquer instante;

2. Nenhuma das mensagens entregues nas listas antigas são perdidas.

Especificamente, uma lista torna-se válida se ela passa por três testes:

- *Teste de Maioria*: requer que uma lista válida tenha a maioria dos processos do grupo. Durante a reforma, um processo pode unir-se a apenas uma lista. O teste de maioria é necessário para assegurar que apenas uma lista válida será formada.
- *Teste de Seqüência*: requer que um processo una-se apenas a uma lista com um número de versão maior que o da lista a qual ele pertencia.
- *Teste de Resiliência*: assegura que nenhuma das mensagens entregues pelas listas antigas são perdidas. Em um sistema *L-resiliente*, uma mensagem entregue deve ser recebida por L processos além do processo que a reconheceu; assim, pelo menos $L + 1$ processos receberam a mensagem. Se a nova lista consiste de um dos $L + 1$ processos que atribuiu o maior timestamp conhecido da lista de processos antiga, então nenhuma das mensagens entregues será perdida.

O protocolo de reforma é um algoritmo de três fases. Na Fase I, o iniciador forma uma nova lista. Ele escolhe o número de versão da nova lista como sendo o número da lista anterior mais 1 e envia uma mensagem para todos os processos pedindo que eles se unam a esta nova lista. Quando um processo concorda em se unir a uma lista, ele então informa ao iniciador o próximo número seqüencial que teria que ser atribuído a uma mensagem na lista antiga, juntamente com o número de versão da lista antiga. O iniciador calcula o maior número de versão mais 1 e usa este número na próxima vez que ele tentar formar uma lista. A combinação do teste de maioria com o de seqüência assegura que todas as lista válidas têm números de versão crescentes. Isto acontece porque quaisquer duas listas válidas devem ter algum processo em comum, o que garante que a nova lista válida sempre tem um número de versão maior que o da lista válida anterior. Quando cada processo ou respondeu ao pedido ou é detectado que ele falhou, então uma nova lista de processos está formada. Os três testes são aplicados a esta nova lista e se ela não for válida, o iniciador aborta a reforma informando aos outros tal ação. Se a nova lista é válida, ele informa aos outros processos esta nova lista.

Na Fase II, a nova lista está formada. Um novo nó ficha é eleito e o número de seqüência inicial das mensagens da nova lista é determinado. O novo nó ficha tem que possuir todas as mensagens trocadas na lista anterior.

Na Fase III, o iniciador gera uma nova ficha e passa-a ao novo nó ficha. Este aceita a ficha e inicia os reconhecimentos das novas difusões, finalizando o procedimento de reforma.

3.1.1 Corretude

A determinação dos números de seqüência, monotonicamente crescentes, pelo nó ficha como reconhecimento das mensagens difundidas garante que todos os processos pertencentes ao grupo ordenam as mensagens de difusão na mesma seqüência. A estratégia de reconhecimento positivo

garante que uma mensagem de difusão será reconhecida enquanto todos os processos do grupo possam se comunicar. A estratégia de passagem da ficha força com que todos os processos recuperem as mensagens perdidas. A estratégia de retransmissão garante que os processos possam recuperar mensagens perdidas. Assim, durante a fase normal do protocolo, todos os processos operacionais receberão e entregarão as mensagens de difusão na mesma ordem.

3.1.2 Desempenho

Dependendo da utilização, o protocolo adaptativamente permuta entre 1 e $L+1$ mensagens de controle por difusão de mensagem. Quando o sistema está trocando muitas mensagens de difusão, o nó ficha agrupa vários reconhecimentos em uma só mensagem e ainda informa quem será o próximo nó ficha. Nesta situação, podemos considerar que é gasto apenas 1 mensagem de controle por cada difusão.

Entretanto, quando o sistema está ocioso, o nó ficha tem que passar a ficha L vezes, para que uma determinada mensagem possa ser entregue. E, assim, ele o faz gerando mensagens *NULL* que repassam a ficha, quando há um *timeout* sem difusões. Assim, o requisito de resiliência L apenas aumenta o número de mensagens transmitidas quando o sistema não está ocupado. Esta é uma característica geral deste protocolo. Há menos mensagens transmitidas por cada difusão quando o sistema está ocupado do que quando está ocioso.

3.1.3 Exemplo

Na Figura 3.2 vemos um exemplo de operação do protocolo com quatro processos, A, B, C e D. O triângulo representa o estado em que todos os processos sabem quem é o nó ficha. Inicialmente, C é o nó ficha. A realiza a difusão de uma mensagem $m(A,2)$ que é recebida por todos exceto D. C reconhece esta mensagem difundindo o seu timestamp 7 determinado pela ordem total — $ACK(7, m(A,2), B)$ e, ao mesmo tempo, indica que está passando a ficha para B (linha pontilhada). Neste instante, D detecta que não recebeu a mensagem com timestamp 7. D difunde, então, um pedido de retransmissão de tal mensagem perdida — "$m(7)$?". Este pedido de retransmissão é recebido por B, o novo nó ficha, que retransmite a mensagem e assim, indica a C que ele aceita a ficha. Em seguida, D difunde a mensagem $m(D,5)$, que é reconhecida por B em $ACK(8, m(D,5), A)$. Se $L = 2$, após estes dois repasses da ficha, as mensagens $m(A,2)$ e $m(D,5)$ podem ser entregues em ordem total.

3.2 Totem

O sistema *Totem* (de "**Tot***al order and* **tem***poral predictability*") [AC93, Aga94, AC95, BLP96] também usa um anel lógico com ficha para fornecer difusões confiáveis em grupos de comunicação. A ficha circula no anel como uma mensagem ponto-a-ponto e, diferentemente do modelo de Chang e Maxemchuk, um processo deve possuir a ficha para poder difundir uma mensagem para os outros processos no anel.

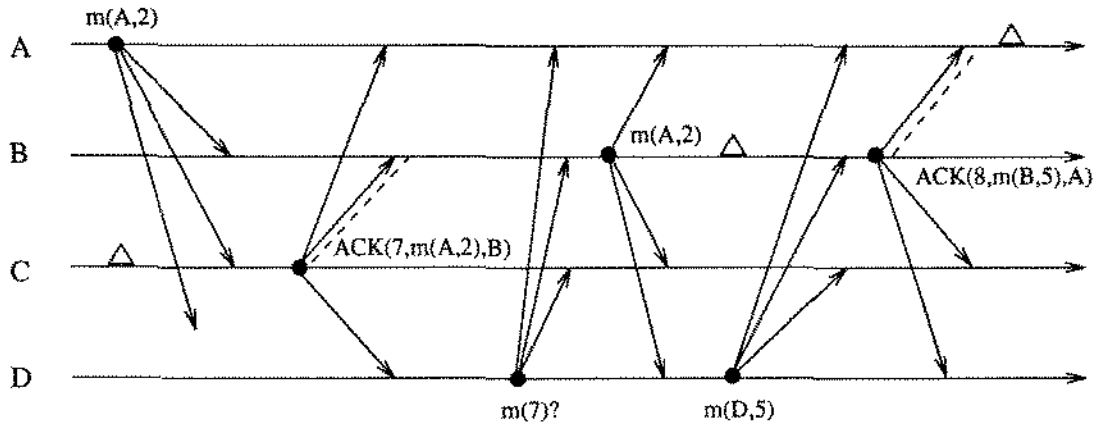


Figura 3.2: Exemplo de operação do protocolo de Chang e Maxemchuk.

O sistema Totem fornece dois serviços de entrega de mensagens totalmente ordenadas, chamados de *concordado* e *seguro*:

- *Entrega Concordada* garante que, quando um processo entrega uma mensagem, ele já entregou todas as mensagens anteriores a esta originadas pelos outros processos.
- *Entrega Segura* garante que, antes de um processo entregar uma mensagem, ele determina que todos os outros processos já receberam esta mensagem.

Assim como no modelo de Chang e Maxemchuk, estes dois serviços entregam mensagens em ordem total de forma que obedecem a ordem causal introduzida por Lamport [Lam78]. Cada mensagem difundida contém um número de seqüência, *timestamp*, derivado do campo *seq* da ficha. Este número de seqüência da ficha é incrementado cada vez que uma mensagem é difundida, e a ficha passa de processo a processo enquanto circula no anel, fornecendo, desta forma, uma simples seqüência de números monotonicamente crescente para as mensagens difundidas no anel.

Estruturas de dados

As mensagens de difusão contém os seguintes campos:

- *sender_id*: Identificação do processo que originou a mensagem.
- *ring_id*: Identificação do anel no qual a mensagem foi originada.
- *seq*: Timestamp da mensagem.
- *contents*: Conteúdo da mensagem.

Os campos *ring_id* e *seq* constituem a identificação da mensagem.

Para difundir uma mensagem, um processo deve possuir a ficha que contém as seguintes informações:

- *ring_id*: Identificação do anel no qual a ficha está circulando.
- *token_seq*: Um número de seqüência que permite identificar cópias redundantes da ficha.
- *seq*: O maior timestamp de qualquer mensagem difundida no anel, ou seja, o timestamp da última mensagem difundida.
- *aru* (*all-received-up-to*): um número de seqüência usado para determinar quais mensagens os processos no anel já receberam, isto é, todos os processos no anel receberam todas as mensagens até a mensagem com este número.
- *aru_id*: A identificação do processo que atualizou o campo *aru* para um valor menor que o de *seq*. Este campo é usado para determinar o processo que não recebeu alguma mensagem difundida.
- *rtr*: Uma lista de pedidos de retransmissão.

O campo *seq* fornece uma ordenação total simples das mensagens para todos os processos no anel. O campo *aru* é o mecanismo de reconhecimento que determina se uma mensagem pode ser entregue no nível *seguro*.

Os processos mantém as seguintes variáveis locais:

- *my_aru*: O timestamp da última mensagem recebida, de forma que o processo já recebeu todas as mensagens com o timestamp menor ou igual a este.
- *my_token_seq*: Identificação da última ficha que ele mantinha.
- *my_high_delivered*: O timestamp da mais recente mensagem entregue.
- *Qn*: Fila de mensagens de difusão a serem enviadas.
- *Qr*: Fila de mensagens recebidas.

my_aru é atualizada pelo processo quando ele recebe as difusões de mensagens. *my_token_seq* é atualizada pelo processo quando ele recebe a ficha, e *my_high_delivered* é atualizada quando ele entrega as mensagens.

O Algoritmo

Na recepção da ficha, um processo difunde as mensagens que foram requisitadas para retransmissão e todas as mensagens em Qn, atualiza a ficha e a envia para o próximo do anel. Para cada mensagem nova que ele difunde, o processo incrementa o campo *seq* da ficha e atribui este timestamp ao valor do campo *seq* da mensagem.

Cada vez que o processo recebe a ficha, ele compara o campo *aru* da ficha com a variável *my_aru* e, se *my_aru* é menor, então ele atribui *aru* ao valor de *my_aru* e *aru_id* ao seu identificador. Se o processo recebe a ficha com o campo *aru_id* igual ao seu identificador, ele então atribui *aru* a *my_aru*. Se *seq* e *aru* são iguais, então ele incrementa *aru* e *my_aru* para *seq* e atribui *aru_id* a -1 (um identificador de processo inválido).

Se o campo *seq* da ficha é maior que *my_aru*, então há difusões de mensagens que este processo não recebeu e, em conseqüência disto, ele insere a identificação destas mensagens no campo *rtr* da ficha. Se o processo recebeu mensagens que aparecem neste campo *rtr*, então ele retransmite estas mensagens removendo os timestamps delas do campo *rtr*. O pseudocódigo executado por um processo ao receber a ficha é visto no Algoritmo 3.2.

Se um processo recebeu uma mensagem m e recebeu e entregou todas as mensagens com o timestamp menor que o da mensagem m e o remetente de m requisitou uma entrega concordada, então o processo pode entregar m na ordem *concordada*. Se um processo pode entregar m na ordem *concordada* e se em duas rotações sucessivas, ele passou a ficha com o valor do campo *aru* maior ou igual ao timestamp de m, então m pode ser entregue pelo processo na ordem *segura*. Quando uma mensagem torna-se segura, não há mais necessidade de retê-la para futuras retransmissões.

O algoritmo de ordenação total é incapaz de continuar quando a ficha é perdida. Assim, um mecanismo de retransmissão da ficha foi incorporado para reduzir a probabilidade de que a ficha seja perdida. Cada vez que um processo repassa a ficha, ele inicia o temporizador de retransmissão da ficha. Este temporizador é cancelado quando um processo recebe ou uma mensagem de difusão ou a ficha. A recepção de uma mensagem de difusão indica que a ficha não foi perdida. Se o temporizador "alarmar", o processo retransmite a ficha para o próximo do anel e reinicia este temporizador.

O campo *token_seq* possibilita o reconhecimento de fichas redundantes. Um processo aceita a ficha somente se o campo *token_seq* da ficha é maior ou igual a *my_token_seq*, caso contrário, a ficha é descartada como redundante. Se a ficha é aceita, o processo incrementa *token_seq* e atribui o seu novo valor a *my_token_set*.

3.2.1 Corretude

Para demonstrar que um protocolo é correto, isto é, ordena e entrega todas as mensagens difundidas pelo grupo, ele precisa obedecer a duas propriedades. A *propriedade de segurança* assegura que a ordem causal de entrega das mensagens nunca será violada, e a *propriedade de entrega* garante que cada mensagem difundida dentro do grupo é em algum instante entregue por cada processo do grupo.

Envio:
Ao receber a ficha t:
se $t.token\_seq < my\_token\_seq$ então 1
descarte t; 2
senão
cancele *temporizador de retransmissão da ficha*; 3
difunda as mensagens de $t.rtr$ que estão em Qr; 4
atualize $t.rtr$ retirando as mensagens retransmitidas; 5
para cada mensagem m em Qn faça 6
$t.seq := t.seq + 1$; 7
$m.seq := t.seq$; 8
difunda m; 9
fim_para
atualize my_aru com base nas mensagens já difundidas; 10
se $(my\_aru < t.aru) \vee (my\_id = t.aru\_id) \vee (t.aru\_id = -1)$ então 11
$t.aru := my\_aru$; 12
se $t.aru = t.seq$ então 13
$token.aru\_id := -1$; 14
senão
$t.aru\_id := my\_id$; 15
fim_se
fim_se
atualize $t.rtr$ inserindo a mensagem m tal que,
$my\_aru < m.seq \leq t.seq$ e $m \notin Qr$; 16
$t.token\_seq := t.token\_seq + 1$; 17
$my\_token\_seq := t.token\_seq$; 18
repasse a ficha; 19
inicie *temporizador de retransmissão da ficha*; 20
entregue as mensagens que satisfazem os critérios de entrega; 21
fim_se

Recebimento e entrega:
Ao receber uma mensagem de difusão:
adicione a mensagem a Qr; 22
atualize a lista de pedidos de retransmissão, caso necessário; 23
atualize my_aru; 24
entregue as mensagens que satisfazem aos critérios de entrega; 25

Algoritmo 3.2: Protocolo de difusão do Totem.

Propriedade de segurança

Na prova da propriedade de segurança, precisamos analisar os dois tipos de entrega do protocolo Totem.

- **Entrega em Ordem Concordada**: A ordenação para entrega das mensagens é Total, e se o processo p entrega a mensagem m_2, e m_1 é qualquer mensagem que precede m_2 na ordem de entrega, então p entrega m_1 antes de entregar m_2.

  As mensagens têm números de seqüência (timestamps) únicos, que como um sub-conjunto de números inteiros não-negativos, formam uma ordem total. Se m_1 é qualquer mensagem que precede m_2, então o timestamp de m_1 é no máximo igual ao timestamp de m_2. Pelo algoritmo, todo processo entrega as mensagens em ordem pelo seu timestamp e não entregaria m_2 até que ele tenha entregue todas as mensagens com timestamp menor.

- **Entrega em Ordem Segura**: Se um processo entrega uma mensagem m e o remetente de m requisitou entrega segura, então este processo determinou que todos os outros processos do grupo receberam e entregarão m.

  Para entregar m em ordem segura o processo p deve repassar a ficha em duas rotações consecutivas com o campo *aru* pelo menos igual ao número de seqüência de m. Assim, p determina que, quando um outro processo q repassou a ficha na primeiras destas rotações, o campo *my_aru* de q estava pelo menos igual ao número de seqüência de m. Conseqüentemente, p determina que, se a ficha não é perdida, então q receberá a ficha na segunda rotação com *aru* pelo menos igual ao número de seqüência de m e poderá então entregar m.

  Além disto, p determina que, se a ficha é perdida, então o primeiro *gap* na seqüência de mensagens recebidas por q deve corresponder a um número de seqüência maior que o de m e, assim, q entregará m antes de se instalar uma nova configuração do anel.

Propriedade de entrega

O mecanismo de retransmissão de mensagens, através do campo *rtr* da ficha, faz com que cada processo que receba a ficha re-envie as mensagens que foram perdidas por alguns processos. Assumindo que as mensagens possam chegar aos seus destinos, em algum instante cada processo terá todas as mensagens difundidas e poderá entregá-las obedecendo aos requisitos da propriedade de segurança.

3.2.2 Desempenho

A latência para ordenar uma mensagem no Totem é uma função do tempo de rotação da ficha. Sob baixa carga, a latência para uma entrega concordada é aproximadamente metade do tempo de rotação da ficha e a latência para uma entrega segura é aproximadamente duas vezes o tempo de rotação da ficha. Assumindo que não há perda de mensagens, o tempo de rotação da ficha é calculado da seguinte forma, usando as seguintes notações:

n Número de processos no anel

T Tempo de rotação da ficha

ρ Utilização do meio de comunicação

a Tempo médio para difundir uma mensagem

r Razão entre o tempo médio para processar e difundir uma mensagem e o tempo médio para apenas difundir

b Tempo de processamento e transmissão da ficha para um processo

m Número médio de difusões de mensagens por um processo durante uma visita da ficha

M Número máximo de mensagens que todos os processos podem difundir em uma rotação da ficha

No cálculo é assumido que o tempo para receber e processar uma mensagem é aproximadamente igual ao tempo para processar e difundir a mensagem. Se o tempo de recebimento é substancialmente maior que o tempo de difusão, então a e r deverão ser derivados do tempo de receber mensagens ao invés do tempo de difundir mensagens.

A utilização útil do meio de comunicação é dado por

$$\rho = \frac{nma}{nb + nmra} = \frac{ma}{b + mra}$$

de onde temos que

$$a = \frac{b\rho}{m(1 - r\rho)}$$

Assim, o tempo de rotação da ficha é

$$T = nb + nmra = \frac{nb}{1 - r\rho}$$

Além disto, o tempo máximo de rotação da ficha é $nb + Mra$, enquanto que a utilização média do meio de comunicação é $Ma/(nb + Mra)$.

3.2.3 Exemplo

Na Figura 3.3 vemos um exemplo de operação do protocolo com três processos, A, B e C. a_i, b_i e c_i representam eventos destes três processos, respectivamente, em que as variáveis locais Q_r e my_aru, e as variáveis da ficha seq, aru, aru_id e rtr são atualizadas. As tabelas abaixo da figura mostram os valores destas variáveis para cada atualização.

As setas pontilhadas representam difusões de mensagens e as setas sólidas passagem da ficha. Uma seta que não toca a linha (horizontal) de execução de um processo indica que a mensagem não chegou a seu destino, ou seja, foi perdida. Assim, a difusão de m_1 por A não é recebida por C, assim como, a difusão de m_2 não é recebida por B. Já m_3 é recebida por todos.

Quando B recebe a ficha, ele difunde as mensagens m_4 e m_5, insere no campo rtr a identificação da mensagem m_2 e repassa a ficha para C. C retransmite m_2, difunde m_6 e insere m_1 e m_5 em rtr. A, ao receber a ficha novamente, retransmite m_1 e m_5 e difunde m_7. No evento c_7 todos os processos receberam todas as mensagens difundidas, e assim, suas variáveis my_aru e o campo da ficha aru são iguais.

3.3 Protocolos derivados

Nesta seção descrevemos alguns protocolos que também utilizam as mesmas estratégias de difusão dos protocolos descritos anteriormente, mudando apenas alguns aspectos. O protocolo TPM assemelha-se ao protocolo Totem, o protocolo de difusão do Sistema Operacional Amoeba usa a mesma idéia de processo centralizador de reconhecimentos do modelo de Chang e Maxemchuk.

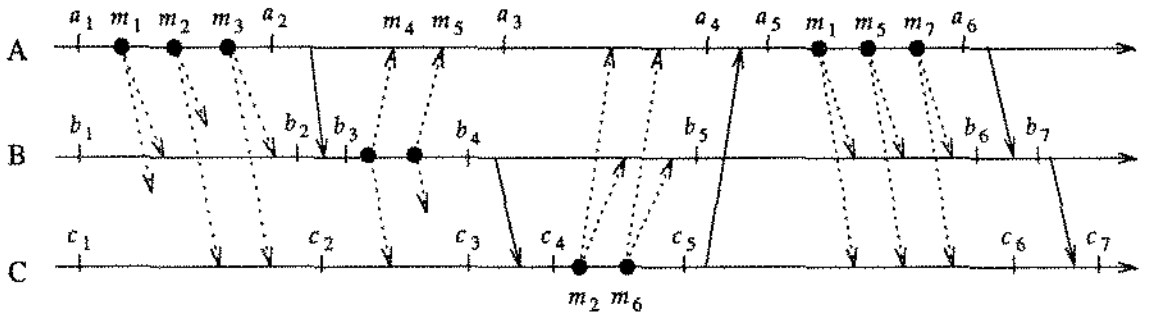
3.3.1 TPM

Assim como o Totem, o protocolo TPM (*Token-Passing Multicast Protocol*) [RM89] usa uma ficha para fornecer difusão confiável ordenada dentro de um grupo de processos, e onde um processo somente pode difundir uma mensagem se ele possui a ficha. Para cada mensagem é dado um número de seqüência derivado da ficha. A ficha é circulada entre os processos do grupo para serializar as transmissões de mensagens. A ficha contém o próximo número de seqüência a ser estampado na próxima difusão. O TPM inicia circulando a ficha para difundir um conjunto de mensagens. Em seguida, a ficha é usada para retransmitir as mensagens pertencentes a este conjunto e que estão faltando para alguns processos do grupo. Quando nenhuma mensagem está faltando por qualquer um dos processos, o conjunto inteiro de mensagens é entregue à aplicação e um novo conjunto de mensagens pode ser introduzido.

O TPM também fornece um algoritmo de gerenciamento dos membros do grupo e de regeneração da ficha. Se a rede é particionada por alguma falha, o componente com a maioria dos membros é escolhido para dar continuidade ao protocolo. A regeneração da ficha é feita através de tentativas de passar a ficha para os membros da lista de processos antiga até que um processo aceite a ficha. A regeneração da ficha é completada somente se a nova lista de processos tem a maioria dos membros do grupo.

3.3.2 O Sistema Amoeba

No sistema operacional Amoeba [HB89, KT91, KT92], as mensagens são enviadas ponto-a-ponto para um coordenador central que se encarrega de atribuir às mensagens um número de seqüência e difundí-las para os outros membros do grupo. Este coordenador reconhece a recepção da mensagem colocando nela o maior número de seqüência, sem intervalos, recebido. Cada grupo de difusão tem seu coordenador central independente. O sistema Amoeba não ordena mensagens entre diferentes grupos.



	Processo A					
Eventos →	a_1	a_2	a_3	a_4	a_5	a_6
Q_r	ϕ	m_1, m_2, m_3	m_1, m_2, m_3, m_4, m_5	$m_1, m_2, m_3, m_4, m_5, m_6$	$m_1, m_2, m_3, m_4, m_5, m_6$	$m_1, m_2, m_3, m_4, m_5, m_6, m_7$
my_aru	0	3	5	6	6	7
token.seq	0	3			6	7
token.aru	0	3			0	0
token.aru_id	-1	-1			C	C
token.rtr	ϕ	ϕ			m_1, m_2	ϕ

	Processo B						
Eventos →	b_1	b_2	b_3	b_4	b_5	b_6	b_7
Q_r	ϕ	m_1, m_3	m_1, m_3	m_1, m_3, m_4, m_5	$m_1, m_2, m_3, m_4, m_5, m_6$	$m_1, m_2, m_3, m_4, m_5, m_6, m_7$	$m_1, m_2, m_3, m_4, m_5, m_6, m_7$
my_aru	0	1	1	1	6	7	7
token.seq			3	5			7
token.aru			3	1			0
token.aru_id			-1	B			C
token.rtr			ϕ	m_2			ϕ

	Processo C						
Eventos →	c_1	c_2	c_3	c_4	c_5	c_6	c_7
Q_r	ϕ	m_2, m_3	m_2, m_3, m_4	m_2, m_3, m_4	m_2, m_3, m_4, m_6	$m_1, m_2, m_3, m_4, m_5, m_6, m_7$	$m_1, m_2, m_3, m_4, m_5, m_6, m_7$
my_aru	0	0	0	0	4	7	7
token.seq				5	6		7
man *token.aru*				1	0		7
token.aru_id				B	C		-1
token.rtr				m_2	m_1, m_5		0

Figura 3.3: Exemplo de operação do protocolo do Sistema Totem.

Na abordagem do Amoeba, cada difusão requer no mínimo uma mensagem ponto-a-ponto e uma mensagem de *broadcast* para cada difusão de mensagem. Ele reduz os requisitos de armazenamento do sistema já que o coordenador é o único computador responsável por manter cópias das mensagens até que a difusão se torne estável. Entretanto, esta centralização pode resultar nos problemas já discutidos anteriormente, como, por exemplo, falha do processo centralizador parar todo o mecanismo de difusão do sistema.

3.4 Comentários

O modelo de difusão do protocolo de Chang e Maxemchuk apresenta algumas vantagens sobre o protocolo do sistema Totem. A principal delas é a velocidade na entrega das mensagens, visto que as difusões simultâneas permitidas pelo protocolo com os respectivos reconhecimentos para estas múltiplas mensagens acelera a validação e a entrega de um conjunto de mensagens "concorrentes" na ordem total.

Como vantagem do modelo do Totem sobre o de Chang e Maxemchuk, podemos citar o controle do fluxo das difusões, que possibilita um mecanismo mais eficiente na recuperação e entrega de mensagens perdidas.

O Totem fornece ainda serviços de difusão e gerenciamento de grupo entre uma coleção de redes locais. O sistema Totem é composto de uma hierarquia de três protocolos. Na camada inferior está o Protocolo de Anel Único (*Single-Ring Protocol*), descrito neste Capítulo e que fornece difusão confiável dentro de um domínio de *broadcast*. Na camada seguinte, o Protocolo de Múltiplos Anéis (*Multiple-Ring Protocol*) fornece difusão confiável e ordenação entre várias redes. *Roteadores* (*Gateways*) são responsáveis pelo repasse das mensagens e pelas mudanças na configuração entre os domínios de broadcast. Cada roteador interconecta dois domínios de broadcast, e participa no Protocolo de Anel Único de cada um deles. Cada domínio pode conter vários roteadores conectando-o a vários outros domínios. Sincronismo virtual estendido foi primeiro implementado no sistema Totem [AC93].

Capítulo 4

Protocolos que utilizam Vetor de Timestamps

Baseados no conceito de Tempo Vetorial [RS96], os protocolos descritos neste capítulo utilizam um vetor de números seqüenciais para marcação do tempo lógico dos eventos (*timestamps*) como principal mecanismo para se alcançar a ordenação das mensagens de difusão. Estes vetores são passados nas difusões e, a partir deles, os processos determinam a ordem lógica das mensagens já difundidas no grupo.

O protocolo CBCAST do sistema ISIS utiliza os vetores de timestamps para alcançar uma ordem causal das mensagens. Já os protocolos de Raynal e Newtop, os utilizam para determinar quais mensagens foram difundidas concorrentemente em uma ordem total de difusões entre diferentes grupos de comunicação.

4.1 ISIS

O sistema de programação distribuída ISIS [BJ87, SS91], e a sua mais recente versão – Horus [BM96, HK95], tem como objetivo principal fornecer uma API ("*Application Program Interface*") para o desenvolvimento de aplicações distribuídas que necessitem de ordenação de mensagens dentro de um grupo de processos. O sistema ISIS fornece dois protocolos para a difusão de mensagens, através de duas primitivas de comunicação: CBCAST para ordenação causal de mensagens, e ABCAST para ordenação total de mensagens. Uma estratégia de vetor de timestamps é usada para assegurar ordenação causal, e um seqüenciador baseado em ficha, similar àquele de Chang e Maxemchuk, é usado para fornecer ordenação total.

O ISIS introduziu o importante conceito de *sincronismo virtual* no qual as mensagens de reconfiguração do grupo são ordenadas relativas às outras mensagens permitindo que uma visão consistente do grupo seja mantida enquanto o sistema muda dinamicamente. Partição e re-junção de rede, assim como recuperação de processos não são suportadas. A descrição do conceito de *sincronismo virtual estendido*, uma versão melhorada com relação ao modelo inicial, é descrito no Capítulo 2, seção 2.5.

4.1.1 Vetor de Timestamps

Os protocolos do ISIS estão baseados em um tipo de relógio lógico, chamado de tempo vetorial (discutido em [RS96]). Um vetor de timestamps para um processo p_i, denotado por $VT(p_i)$, é um vetor de tamanho n (onde n é o número de processos no grupo), indexado pelo número do processo.

1. Quando p_i inicia sua execução, $VT(p_i)$ é iniciado com zeros.
2. Para cada evento $envia(m)$ em p_i, $VT(p_i)[i]$ é incrementado de 1.
3. Cada mensagem de difusão pelo processo p_i é estampada com o valor incrementado de $VT(p_i)$.
4. Quando o processo p_j recebe a mensagem m de p_i contendo $VT(m)$, p_j modifica seu vetor de timestamps da seguinte maneira:

$$\forall k \in 1..n : VT(p_j)[k] = max(VT(p_j)[k], VT(m)[k])$$

Isto é, o vetor de timestamps atribuído à mensagem m conta o número de mensagens por remetente que causalmente precedem m. As regras para comparar vetores de timestamps são:

1. $VT_1 \leq VT_2$ se e somente se $\forall i : VT_1[i] \leq VT_2[i]$
2. $VT_1 < VT_2$ se $VT_1 \leq VT_2$ e $\exists i : VT_1[i] < VT_2[i]$

Assim, dado duas mensagens m e m', $m \rightarrow m'$ se e somente se $VT(m) < VT(m')$, ou seja, os vetores de timestamps representam causalidade precisamente.

4.1.2 O protocolo CBCAST

Suponha que um conjunto de processos P comunicam-se entre si usando apenas difusões para o conjunto inteiro de processos no sistema; isto é, $\forall m : dests(m) = P$. O protocolo aqui descrito assume que não há falhas na comunicação e que os processos não falham e, assim, garante que cada processo p recebe as mensagens enviadas para ele, retarda-as se necessário, inserindo-as em uma fila Qr, e as entrega em uma ordem consistente com a causalidade:

$$m \rightarrow m' \Rightarrow \forall p : entrega_p(m) \rightarrow entrega_p(m')$$

A idéia principal é rotular cada mensagem com um timestamp, $VT(m)[k]$, indicando precisamente quantas difusões pelo processo p_k precedem m. Um receptor de m retardará m até que $VT(m)[k]$ mensagens tenham sido entregues por p_k.

O protocolo é como segue:

(1) Antes de enviar m, o processo p_i incrementa $VT(p_i)[i]$ de 1 e envia o vetor $VT(m) = VT(p_i)$ na mensagem.

(2) Na recepção da mensagem m enviada por p_i e estampada com $VT(m)$, o processo $p_j \neq p_i$ atrasa a entrega de m até que:

$$\forall k : 1..n \begin{cases} VT(m)[k] = VT(p_j)[k] + 1 & \text{se } k = i \\ VT(m)[k] \leq VT(p_j)[k] & \text{se } k \neq i \end{cases}$$

O processo p_j não necessita retardar mensagens recebidas dele mesmo. Mensagens retardadas são mantidas em uma fila, chamada *delay_queue*.

(3) Quando uma mensagem m é entregue, $VT(p_j)$ é atualizado de acordo com o protocolo da seção 4.1.1.

O passo (2) é a chave do protocolo. Ele garante que qualquer mensagem m' transmitida causalmente antes de m (e daí com $VT(m') < VT(m)$) será entregue em p_j antes de m. O pseudocódigo deste protocolo é visto no Algoritmo 4.1.

Exemplo

Um exemplo no qual esta regra é usada para atrasar a entrega de uma mensagem é visto na Figura 4.1.

No evento e_1 de recepção da mensagem m_2, p_3 detecta que não recebeu ainda a mensagem com timestamp 1 de p_1, colocando m_2 em Qr. Após o recebimento e entrega de m_1 (evento e_2), p_3 finalmente entrega m_2 em ordem causal (evento e_3), já que $m_1 \rightarrow m_2$. A seta pontilhada representa o atraso na entrega de m_2.

Corretude

A corretude do protocolo CBCAST é provada em dois estágios: primeiro, mostrando que a causalidade nunca é violada (propriedade de segurança) e segundo, demonstrando que o protocolo nunca atrasa a entrega de uma mensagem indefinidamente (propriedade de entrega).

Propriedade de Segurança

Considere as ações de um processo p_j que recebe duas mensagens m_1 e m_2 tal que $m_1 \rightarrow m_2$.

- *Caso 1.* m_1 e m_2 são transmitidas pelo mesmo processo p_i. Assumindo que não há falhas na comunicação, p_j, em um determinado momento, recebe tanto m_1 quanto m_2. Por construção, $VT(m_1) < VT(m_2)$, daí pela regra (2), m_2 pode ser entregue somente depois que m_1 for entregue.

- *Caso 2.* m_1 e m_2 são transmitidas por dois processos p_i e $p_{i'}$. Mostraremos por indução nas mensagens recebidas por p_j que m_2 não pode ser entregue antes de m_1. Assuma que m_1 não foi entregue e que p_j recebeu k mensagens.

Envio:

Quando p_i **envia** m **para** P

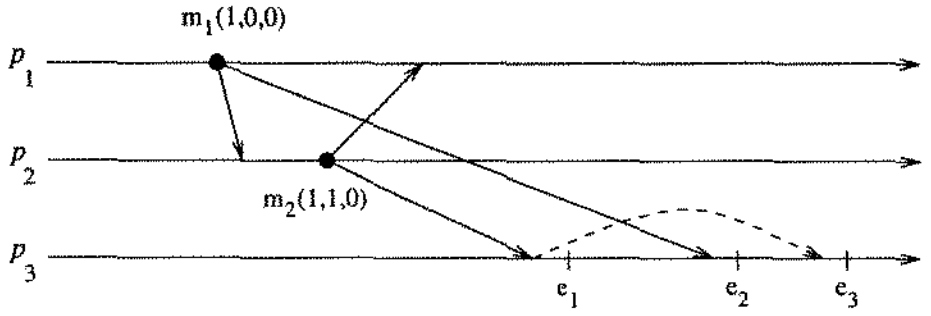
$VT(p_i)[i] := VT(p_i)[i] + 1;$ 1
$VT(m) := VT(p_i);$ 2
divulgue $m(VT(m))$; 3

Recebimento e Entrega:

Quando $m(VT(m))$ **chega em** p_j **enviada por** p_i

entrega := VERDADEIRO; 4
para $k = 1$ até n faça 5
se $(k = i)$ então 6
se $(VT(m)[k] \neq VT(p_j)[k] + 1)$ então 7
entrega := FALSO; 8
$k := n$; { *sai do laço* } 9
fim_se
senão
se $(VT(m)[k] > VT(p_j)[k]$ então 10
entrega := FALSO; 11
$k := n$; { *sai do laço* } 12
fim_se
fim_se
fim_para
se *entrega* então 13
para $k = 1$ até n faça 14
$(VT(p_j)[k] := max(VT(p_j)[k], VT(m)[k]);$ 15
fim_para
entregue m; 16
re-execute este procedimento para $\forall m \in Qr$; 17
senão
insira m em Qr; 18
fim_se

Algoritmo 4.1: Protocolo CBCAST.



$j = 3$	Eventos		
	e_1	e_2	e_3
Valor de i	2	1	2
$VT(m)$	(1,0,0)	(1,0,0)	(1,1,0)
$VT(p_3)$	(0,0,0)	(1,0,0)	(1,1,0)
Qr	$m_2(1,0,0)$	$m_2(1,1,0)$	ϕ
Entrega		$m_1(1,0,0)$	$m_2(1,1,0)$

Figura 4.1: Usando a regra de vetores de timestamps para atrasar a entrega de uma mensagem.

Observe primeiro que $m_1 \rightarrow m_2$, e então $VT(m_1) < VT(m_2)$. Em particular, se considerarmos o elemento do vetor correspondente ao processo p_i, o remetente de m_1, teremos:

$$VT(m_1)[i] \leq VT(m_2)[i]. \tag{4.1}$$

- *Caso básico.* A primeira mensagem entregue por p_j não pode ser m_2. Lembre que se nenhuma mensagem foi entregue a p_j, então $VT(p_j)[i] = 0$. Entretanto, $VT(m_1)[i] > 0$ (pois m_1 é enviada por p_i), portanto $VT(m_2)[i] > 0$. Pela aplicação da regra (2) do protocolo, m_2 não pode ser entregue por p_j.

- *Passo indutivo.* Suponha que p_j recebeu k mensagens, nenhuma das quais é uma mensagem m tal que $m_1 \rightarrow m$. Se m_1 não foi ainda entregue, então

$$VT(p_j)[i] < VT(m_1)[i]. \tag{4.2}$$

Isto procede pois a única maneira de atribuir um valor a $VT(p_j)[i]$ maior que $VT(m_1)[i]$ é entregar uma mensagem de p_i que foi enviada subseqüente a m_1, e tal mensagem seria causalmente dependente de m_1. Das relações 4.1 e 4.2 temos que

$$VT(p_j)[i] < VT(m_2)[i].$$

Pela aplicação do passo 2 do protocolo, as $k + primeira\_mensagem$ entregues por p_j não podem ser m_2.

Propriedade de Entrega

Suponha que existe uma mensagem m enviada pelo processo p_i que pode nunca ser entregue ao processo p_j. O passo 2 implica que ou:

$$\exists k : 1..n \begin{cases} VT(m)[k] \neq VT(p_j)[k] + 1 & \text{para } k = i, \text{ ou} \\ VT(m)[k] > VT(P_j)[k] & k \neq i \end{cases}$$

e que m não foi transmitida pelo processo p_j. Consideraremos estes casos a seguir:

- $VT(m)[i] \neq VT(p_j)[i] + 1$; isto é, m não é a *próxima mensagem* a ser entregue de p_i para p_j. Note que apenas um número finito de mensagens podem preceder m. Desde que todas as mensagens são difusões para todos os processos e os canais de comunicação são livres de perdas e seqüenciais, segue que deve haver alguma mensagem m' enviada por p_i que p_j recebeu anteriormente, ainda não foi entregue e que é a próxima mensagem de p_i, isto é, $VT(m')[i] = VT(p_j)[i] + 1$. Se m' também é retardada, ela deve estar no outro caso.

- $\exists k \neq i : VT(m)[k] > VT(p_j)[k]$. Seja $n = VT(m)[k]$. A n-ésima transmissão do processo p_k, deve ser alguma mensagem $m' \rightarrow m$ que foi ou recebida em p_j ou foi recebida e é retardada. Sob a hipótese de que todas as mensagens são enviadas para todos os processos, m' já foi difundida para p_j. Desde que o sistema de comunicação em um dado instante entrega todas as mensagens, nós podemos assumir que m' foi recebida por p_j. O mesmo raciocínio aplicado a m pode ser aplicado a m'. O número de mensagens que devem ser entregues antes de m é finito e a relação "$>$" é acíclica, então isto leva a uma contradição.

4.1.3 O protocolo ABCAST

O protocolo ABCAST (*Atomic Broadcast*) é uma versão estendida do protocolo CBCAST (*Causal Broadcast*) para implementar ordenação total. A ordem total das mensagens de um grupo é determinada usando um protocolo similar ao de Chang e Maxemchuk. A ficha é passada entre os membros do grupo, e o possuidor atual da ficha impõe a ordem total das mensagens concorrentes na ordem parcial para o grupo. O possuidor da ficha envia uma mensagem indicando o resultado da sua decisão de ordenação para os outros processos.

Algoritmo

Associado com cada grupo de processos g há um processo seqüenciador, chamado de *token holder* (ou possuidor da ficha), encarregado de manter a ficha para o protocolo ABCAST. Uma mensagem ABCAST é unicamente identificada por um timestamp em ordem total, denotado por $uid(m)$.

Para difundir uma mensagem m por ABCAST, o protocolo é realizado nos seguintes estágios. Se o processo remetente de m é o possuidor da ficha, então ele inicia o protocolo no passo 2.

(1) O remetente difunde m usando CBCAST mas marca-a como *não-entregável*. Os processos que não sejam o possuidor da ficha (incluindo o remetente) que recebem esta mensagem

colocam m na fila Qr de retardo do CBCAST da maneira usual, e não removem m desta fila para entrega mesmo depois que todas as mensagens que a precedem tenham sido entregues. Desta forma, um processo pode ter um certo número de mensagens ABCAST atrasadas no início da sua fila Qr. Isto previne a entrega de mensagens CBCAST causalmente subseqüentes, pois o vetor de timestamp não é atualizado enquanto as entregas não ocorrem. Por outro lado, uma mensagem CBCAST que precede ou é concorrente com uma destas mensagens ABCAST não-entregáveis não será atrasada.

(2) O possuidor da ficha trata as mensagens ABCAST dele ou que chegam como se elas fossem mensagens CBCAST, entregando-as da mesma forma como no protocolo CBCAST. Entretanto, ele determina o timestamp *uid* de cada mensagem ABCAST colocando-as em ordem total.

(3) Depois que o processo possuidor da ficha entregou uma ou mais mensagens ABCAST, ele usa o protocolo CBCAST para difundir uma mensagem especial, denominada **sets-order**, que informa a lista de mensagens ABCASTs, recebidas no passo 2, com seus respectivos timestamps *uid*. Se desejado, um novo possuidor da ficha pode ser especificado nesta mensagem.

(4) Ao receber a mensagem **sets-order**, um processo a insere na fila CBCAST de retardo da mesma forma como as outras mensagens. Em um algum momento todas as mensagens ABCAST, referidas pela mensagem **sets-order**, serão recebidas, e todas as mensagens CBCAST que precedem a mensagem **sets-order** terão sido entregues (propriedade de entrega do CBCAST).

Lembre que a relação "$\rightarrow$" determina uma ordem parcial das mensagens na fila de retardo. O protocolo, neste momento, reordena as mensagens ABCAST atuais colocando-as na ordem dada pela mensagem **sets-order**, e as marca como *entregáveis.*

(5) As mensagens ABCAST *entregáveis* são, então, entregues e retiradas do início da fila Qr.

O passo 4 é a chave deste protocolo. Este passo faz com que todos os participantes entreguem as mensagens ABCAST na ordem que o possuidor da ficha determinou. Esta ordem será consistente com a causalidade pois o possuidor da ficha por si próprio trata estas mensagens como CBCAST's.

Como observado no passo 3, a ficha pode ser transferida entre os processos. A principal vantagem disto é que um processo que é a única fonte de mensagens ABCAST pode ser designado como o *token holder*; e neste caso, mensagens de **sets-order** separadas seriam desnecessárias. O Algoritmo 4.2 apresenta o protocolo ABCAST, onde p_i não é possuidor da ficha.

Estruturas de Dados

As mensagens de difusão possuem os seguintes campos:

- *entreg.* Valor: verdadeiro ou falso. Indica se a mensagem pode ou não ser entregue.

- *tipo*. Valor: CBCAST, ABCAST ou SETS-ORDER. Informa o protocolo ou o propósito da mensagem.
- *ts*. Timestamp da mensagem em ordem total.

total é a variável local mantida pelo processo possuidor da ficha que armazena o número de ordem da última mensagem ABCAST difundida. Qr é a fila de mensagens mantidas pelos processos que retardaram as suas entregas.

Envio:

Quando p_i **envia** m **para** P

$m.entreg := \text{FALSO};$	1
$m.tipo := \text{ABCAST};$	2
$VT(p_i)[i] := VT(p_i)[i] + 1;$	3
$VT(m) := VT(p_i);$	4
divulgue $m(VT(m))$;	5

Recebimento e Entrega:

Quando $m(VT(m))$ **chega em** p_j **enviada por** p_i

se $m.tipo = \text{CBCAST}$ então	6
execute CBCAST para processar m;	7
senão se $m.tipo = \text{ABCAST}$ então	8
se p_j possui a ficha então	9
prepare uma mensagem s do tipo SETS-ORDER;	10
enquanto houver mensagem ABCAST m em Qr faça	11
$total := total + 1;$	12
$m.ts := total;$	13
$m.entreg := \text{VERDADEIRO};$	14
insira $m.total$ em s;	15
entregue m;	16
fim_enquanto	
divulgue s;	17
senão	
insira m em Qr;	18
fim_se	
senão se $m.tipo = \text{SETS-ORDER};$	19
entregue as mensagens ABCAST's de Qr na ordem definida por m;	20
fim_se	

Algoritmo 4.2: Protocolo ABCAST.

Exemplo

A Figura 4.2 mostra um exemplo de operação do protocolo ABCAST. p_2 é o possuidor da ficha, e determina a ordem total das mensagens m_1 e m_3, através da mensagem **sets-order**.

$SO(m_3, m_1)$. As setas pontilhadas representam o atraso na entrega das mensagens.

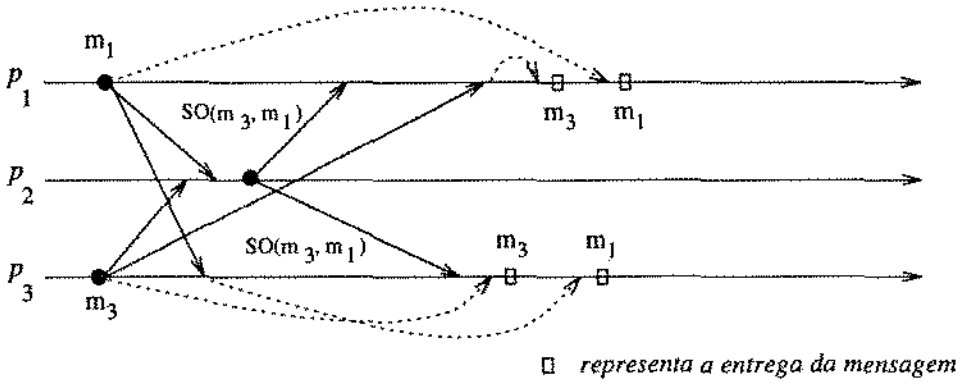


Figura 4.2: Exemplo de operação do protocolo ABCAST.

4.1.4 Desempenho

O número de mensagens por difusão CBCAST é 1, ou seja, é a própria difusão, pois a entrega de uma mensagem em ordem causal é imediata, caso as predecessoras desta mensagem já tenham sido recebidas.

O custo de uma difusão ABCAST depende da localização onde se originam as difusões e da freqüência com a qual a ficha é passada. Se as difusões tendem a se originarem no mesmo processo repetidamente então, uma vez que a ficha é passada para este processo, o custo é um CBCAST por ABCAST. Se elas originam-se aleatoriamente e a ficha não é passada, o custo é $1 + 1/k$ CBCAST's por ABCAST, se assumirmos que uma mensagem **sets-order** é enviada para propósitos de ordenação depois de k ABCAST's.

4.2 Protocolo de Raynal e Mostefaoui

Raynal e Mostefaoui [MR93] propõem um melhoramento ao protocolo CBCAST do Sistema ISIS para difusões causais entre grupos de comunicação. Este melhoramento é alcançado através da redução do número de vetores de timestamps que são passados nas mensagens de difusão.

No protocolo CBCAST do ISIS, se um processo pertence a mais de um grupo, as mensagens difundidas por ele carregam consigo os vetores de timestamps de cada grupo. O protocolo proposto por Raynal e Mostefaoui necessita que se passe nas difusões apenas um vetor contendo os timestamps de cada grupo. Assim, se o número de grupos é grande e cada grupo contém muitos processos, a economia no tamanho das mensagens poderá ser bastante sensível. Uma generalização destes esquemas pode ser vista na Figura 4.3. p_i é um processo que pertence aos grupos $G1$ e $G2$. Para cada grupo x, p_i mantém um vetor timestamp VT_x. No protocolo proposto, as mensagens carregam apenas um vetor com dois elementos, representando os últimos timestamps gerados em cada grupo de p_i.

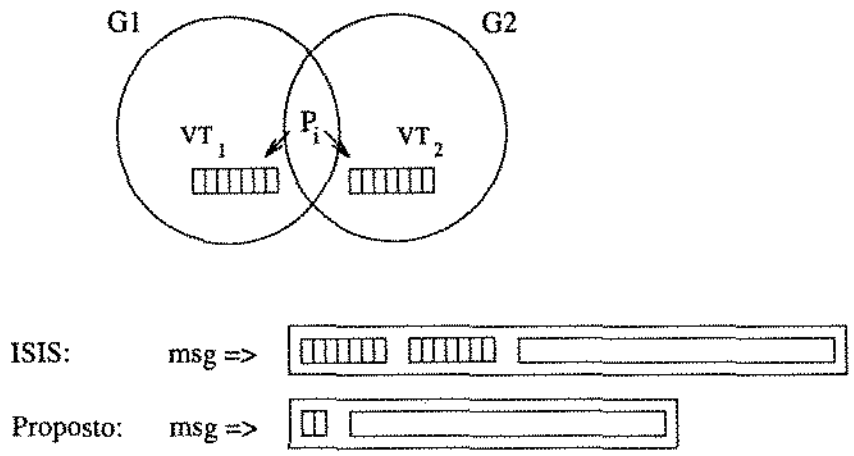


Figura 4.3: Comparação do tamanho das mensagens de difusão

Princípio do protocolo

A comunicação entre os membros de um grupo procede através de fases de sincronização. Em cada fase, cada processo difunde uma mensagem, e uma fase só inicia quando todas as mensagens da fase anterior foram entregues. O princípio do proposto protocolo é criar tais fases seqüenciais de forma que o timestamp das mensagens seja o número de suas fases e, a partir disto, entregar as mensagens de acordo com a ordem destes timestamps.

É possível que um processo p_j não necessite difundir mensagens durante várias fases. Neste caso, tal processo p_j é requisitado a informar quando ele receber uma mensagem com timestamp t. Para isto, p_j deve enviar um tipo especial de mensagem (nula) para sincronização, $resinc(t+1)$, indicando que a próxima fase de difusões não será menor que $t+1$.

Descrição do Protocolo

Seja G o conjunto de todos os grupos pertencentes ao sistema. G_i é o conjunto dos grupos dos quais o processo p_i é membro. Para cada processo p_i, é associado um processo de controle CTL_i que implementa a comunicação em grupo e as regras que asseguram a ordem causal das mensagens. Há um CTL_i para cada grupo de p_i (semelhante à idéia de uma camada de comunicação em grupo (CG) discutida na seção 2.1. Cada CTL_i gerencia um vetor $esperado_i^x$ por grupo g_x ao qual p_i pertence, com uma entrada para cada membro do grupo:

Para cada $p_j \in gx \in G_i : esperado_i^x[j] = \alpha \iff CTL_i$ sabe que o próximo timestamp usado por CTL_j para difundir mensagens dentro de g_x será maior ou igual a α.

A fim de assegurar entregas corretas, cada CTL_i mantém um vetor K_i de tamanho $|G_i|$ (uma entrada por grupo) para atribuir os timestamps das mensagens destinadas a cada grupo. K_i é iniciado para $(0, ..., 0)$. Seu significado é o seguinte:

$K_i[x] = \alpha \iff$ α é o maior valor de timestamp em g_x conhecido por CTL_i. Se $p_i \in g_x$ nós sabemos que $K_i[x] = esperado_i^x[i]$, já que todos os processos enviam mensagens em cada fase.

Uma mensagem de difusão m contém as seguintes informações: identificação do processo remetente, vetor timestamp K deste processo e número do grupo ao qual a mensagem é destinada. A entrega a p_i de uma mensagem m com vetor timestamp K é feita assim que CTL_i esteja certo de que a ordem causal não será violada. Isto é determinado quando:

$\forall g_x \in G_i : \forall p_j \in g_x : esperado_i^x[j] \geq K[x] \Rightarrow$ Ou seja, os timestamps de todos os processos de cada grupo de p_i é pelo menos igual aos timestamps informados na mensagem, assim, não há nenhuma outra mensagem precedente a m que p_i não tenha recebido.

O Algoritmo

Os quatro procedimentos, vistos no Algoritmo 4.3, mostram a operação do protocolo para o envio, recebimento, sincronização e entrega das mensagens de difusão.

Sempre que um elemento de $esperado_i$ é atualizado e a fila de mensagens $pendentes_i$, contendo as mensagens recebidas e ainda não entregues, não está vazia, o procedimento de entrega é executado.

Um aspecto importante neste protocolo é que se o grafo que representa a ligação entre os grupos (interseção de processos) não possui ciclos, então o vetor K_i pode ser reduzido para apenas um valor $K_i[x]$, para todos os grupos $g_x \in G_i$. Em outras palavras, teríamos $K_i[x] = esperado_i^x[i]$, e assim o vetor K_i inteiro não seria mais necessário.

4.2.1 Corretude

Na prova da corretude do protocolo, assume-se que os canais de comunicação são FIFO e confiáveis, ou seja, não há perda de mensagens.

Propriedade de Segurança

Para se provar esta propriedade consideraremos dois casos seguintes:

- *Caso 1:* p_i pode imediatamente entregar uma mensagem m_2 que ele acabou de enviar sem violar a ordem casual (linha 4).

  Considere primeiro $m_1 \rightarrow m_2$ e que tanto m_1 quanto m_2 foram difundidas por p_i, então, de acordo com a regra de entrega, m_1 já foi entregue; segundo, se m_1 foi enviada por outro processo p_j, a definição de $m_1 \rightarrow m_2$ implica que a entrega de m_1 por p_i ocorreu antes do envio de m_2.

- *Caso 2:* Se:

Envio:

Quando p_i **envia** m **para** g_x

divulgue (m, K_i, x) para g_x;	1
$esperado_i^x[i] := esperado_i^x[i] + 1$;	2
$K_i[x] := esperado_i^x[i]$;	3
entregue m para p_i;	4

Recebimento:

Quando (m, K, x) **chega em** p_i **enviada por** p_j

ponha (m, K) na fila de mensagens $pendentes_i$;	
$esperado_i^x[j] := K[x] + 1$;	5
se $esperado_i^x[i] < K[x] + 1$ então	
$esperado_i^x[i] := K[x] + 1$;	6
$K_i[x] := K[x] + 1$;	7
divulgue $resinc(K[x] + 1, x)$ para g_x;	8
fim_se;	

Sincronização:

Quando $resinc(t, x)$ **chega em** p_i **vinda de** p_j

$esperado_i^x[j] := t$;	9

Entrega:

Seja (m, k) **um elemento da fila** $pendentes_i$

se $\forall g_x \in G_i : \forall p_j \in g_x : esperado_i^x[j] \geq K[x]$ então	10
retire (m, K) de $pendentes_i$;	
$\forall g_y \notin G_i : K_i[y] := max(K_i[y], K[y])$;	11
fim_se;	

Algoritmo 4.3: Protocolo de Raynal e Mostefaoui.

- p_i é o processo destino para m_1 e m_2
- $m_1 \rightarrow m_2$
- m_2 chegou e é entregável

então m_1 também chegou em p_i e é entregável (e é então entregue antes de m_2 pois tem um timestamp menor).

Sejam $K(m_1)$ e $K(m_2)$ os timestamps de m_1 e m_2, respectivamente. Se m_2 é entregável, temos que (linha 10):

$$\forall g_x \in G_i : \forall p_j \in g_x : esperado_i^x[j] \geq K(m_2)[x]$$

Como $\forall g_x \in G : K(m_2)[x] \geq K(m_1)[x]$ e as variáveis $esperado_x^i[j]$ nunca são decrementadas, a condição de entrega de m_1 é também verdadeira.

Mostraremos agora que m_1 chegou em p_j. Sejam p_{j1} e g_{x1} o remetente e o grupo destino de m_1. Como $m_1 \rightarrow m_2$, $\forall g_x \in G : K(m_2)[x] \geq K(m_1)[x]$ e $K(m_2)[x_1] > K(m_1)[x_1]$. p_i é o membro de g_{x1} (ele é um processo destino de m_1). Como a condição de entrega de m_2 é verdadeira, nós concluímos que:

$$esperado_i^{x1}[j_1] \geq K(m_2)[x_1] > K(m_1)[x_1].$$

Propriedade de entrega

A propriedade de entrega garante que cada mensagem difundida dentro de um grupo destino é em algum instante entregue por cada processo deste grupo.

Uma mensagem difundida por p_i é entregue por ele apenas uma vez (linha 4). Agora considere o caso de uma mensagem (m, K) que chegou e foi posta em $pendentes_i$. Se a sua condição de entrega é falsa, teremos que (linha 10): $\exists g_x \in G_i, \exists p_j \in g_x : esperado_i^x[j] < K(m)[x]$. Considere tal dupla (g_x, p_j) e os dois casos seguintes (p_i e p_j pertencentes a g_x).

- *Caso 1:* $esperado_j^x[j] \geq K(m)[x]$. Isto significa que CTL_j enviou para g_x uma mensagem m_1 com timestamp $K(m_1)$ com $K(m_1) \geq K(m)[x] - 1$ (linhas 1, 2 e 3) ou $resinc(t, x)$ com $t \geq K(m)[x]$ (linhas 6, 7 e 8 para p_j). Como os canais são confiáveis, os dados e as mensagens de resincronização chegam em seus destinos. Em suas chegadas CTL_i atualiza $esperado_x^x[j]$:
  - para $K(m_1)[x] + 1$ no caso da mensagem m_1 ter sido enviada (linha 5).
  - para t no caso de uma mensagem $resinc$ ter sido enviada (linha 9).

  Em ambos os casos o novo valor de $esperado_i^x[j]$ será maior ou igual a $K(m)[x]$.
- *Caso 2:* $esperado_j^x[j] < K(m)[x]$. A partir da monotonicidade de $esperado_j^x[j]$ e do teste na linha 6, CTL_j ainda não recebeu m. Como os canais são confiáveis, m certamente chegará em CTL_j. Quando m chega, CTL_j atualiza $esperado_j^x[j]$ para $K(m)[x]+1$ (linha 6) e envia $resinc(K(m)[x] + 1, x)$ para CTL_i (linha 8); e assim, nós voltamos ao caso 1.

Assim, para cada dupla (g_x, p_j) teremos, para um determinado momento, em ambos os casos $esperado_i^x[j] \geq K(m)[x]$ e então a condição de entrega para m será verdadeira.

Isto prova que cada mensagem é certamente entregue. Além disto, como uma mensagem entregue é removida de $pendentes_i$, então ela é entregue somente uma vez.

4.2.2 Desempenho

No protocolo CBCAST do ISIS se a condição de entrega de uma mensagem m que chegou em p_i for avaliada como falsa, isto indica que alguma mensagem que foi enviada por p_i e que é causalmente ordenada antes de m não chegou ainda em p_i. Em outras palavras, a entrega de uma mensagem é retardada se, e somente se, alguma mensagem que ainda não chegou é para ser entregue antes desta primeira. Neste sentido, os protocolos do ISIS são ótimos.

O protocolo de Raynal e Mostefaoui pode atrasar a entrega de uma mensagem pela mesma razão, mas possivelmente também pelas mensagens *resinc* que ainda não chegaram. Quando a execução distribuída é síncrona o protocolo é ótimo (pois neste caso mensagens de *resinc* não são necessárias. Quanto mais síncrona a execução for, mais próximo do ótimo serão as ocorrências de entregas.

Como o protocolo proposto é tomado em duas fases, a fase de envio e a de sincronização, e para cada mensagem difusão são geradas $n-1$ mensagens de sincronização, então o desempenho geral do protocolo é n mensagens por difusão, onde n é o número de processos no grupo.

4.2.3 Exemplo

Três processos p_1, p_2 e p_3 são estruturados em três grupos: $g_1 = \{p_1, p_2\}$, $g_2 = \{p_2, p_3\}$ e $g_3 = \{p_1, p_3\}$. Neste caso, o grafo de ligação dos elementos dos grupos é cíclico, o que obriga a que cada participante manter um vetor de timestamps para cada grupo, mesmo para grupos aos quais ele não pertence [MR93]. As mensagens são difundidas da seguinte maneira: m_1 por p_1 dentro de g_1, m_2 por p_1 dentro de g_3 e m_3 por p_3 dentro de g_2, com as seguintes relações causais: $m_1 \rightarrow m_2$ e $m_2 \rightarrow m_3$. Assim, m_1 deve ser entregue a p_2 antes de m_3 a fim de assegurar ordem causal. Na Figura 4.4, K_i aparece como $K_i[g_1, g_2, g_3]$.

4.3 Newtop

Assim como o protocolo de Raynal e Mostefaoui, o protocolo Newtop [ES93, ES94], desenvolvido na Universidade de Newcastle ("**NEW***castle* **T***otal* **O***rder* **P***rotocol*"), também utiliza o conceito de fases de sincronização para prover ordenação de mensagens entre grupos de comunicação. A diferença básica deste modelo para o anterior é que as mensagens de difusão carregam apenas o timestamp da fase à qual ela pertence, e a partir disto os processos dos vários grupos determinam uma **ordem total** das mensagens geradas entre as várias fases.

Neste modelo foi introduzido o conceito de *blocos causais*. Cada bloco representa uma fase do modelo de sincronização. As mensagens com o mesmo número do bloco são classificadas como *concorrentes*, e o Bloco Causal, contendo um conjunto de vetores de timestamps, armazena as

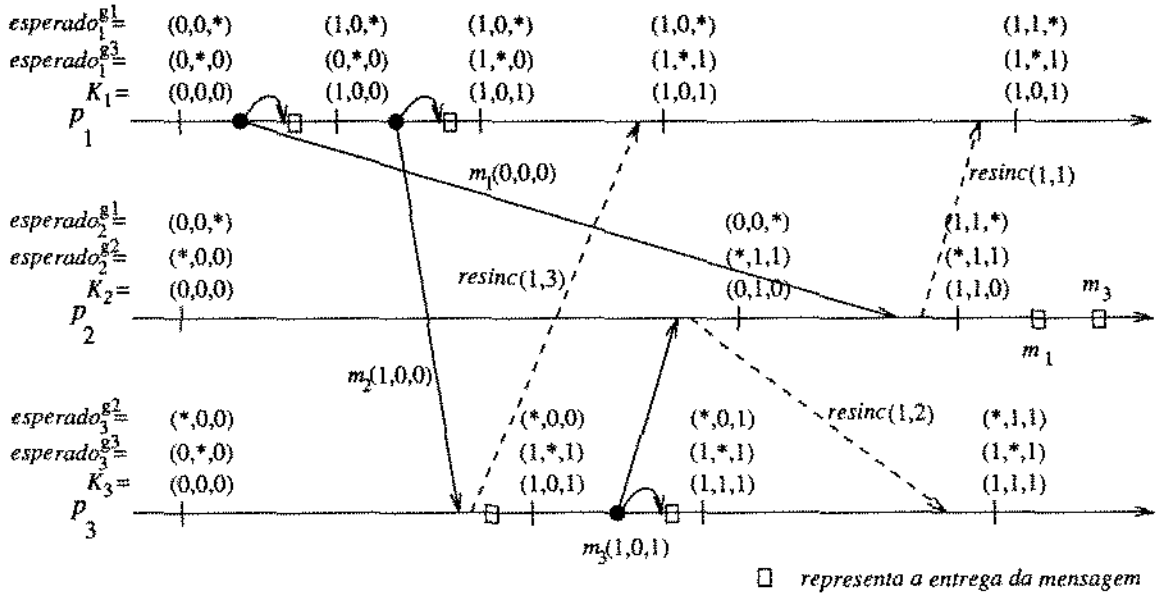


Figura 4.4: Um exemplo de entrega causal entre grupos de comunicação.

informações de ordem destas mensagens concorrentes. Uma vez que um bloco causal torna-se completo, ou seja, não há mais mensagens distintas a serem recebidas com este número de bloco, as mensagens recebidas nesta fase (bloco) são entregues à aplicação em ordem total.

A fim de assegurar que, em determinado instante, um bloco causal estará completo, cada processo transmite uma mensagem nula, se o mesmo não tiver alguma mensagem de difusão a enviar. Isto é equivalente ao método de sincronização de fases visto no modelo de Raynal e Mostefaoui através das mensagens nulas *resinc*.

Descrição do Protocolo

Seja G o conjunto de todos os grupos do sistema. G_i é o conjunto dos grupos dos quais o processo p_i é membro. Cada processo p_i mantém um relógio lógico chamado de Contador de Bloco (*Block Counter*) denotado por BC_i cujo valor (inteiro) crescerá monotonicamente e determinará a ordem total das mensagens.

Uma mensagem de difusão m tem os seguintes campos: $m.b$, número do bloco no qual a mensagem foi gerada; e $m.g$, número do grupo ao qual o remetente a enviou.

BC_i é iniciado com zero e os dois eventos nos quais BC_i pode ser incrementado são os seguintes:

AC1: (*Avança Contador durante envia$_i$*(m)): Antes de difundir m, p_i incrementa BC_i de um, e atribui este novo valor a $m.b$; e

AC2: (*Avança Contador durante entrega$_i$*(m)): Antes de entregar m, p_i fixa $BC_i = max\{BC_i, m.b\}$.

São definidas três propriedades com relação aos números de bloco das mensagens de difusão:

pr1: $envia_i(m) \rightarrow envia_i(m')$ e $m.g = m'.g \Rightarrow m.b < m'.b$;

pr2: Para qualquer m, $p_j \in m.g$: $entrega_j(m) \rightarrow envio_j(m'') \Rightarrow m.b < m''.b$; e

pr3: $\forall m', m''$: $m'.b = m''.b \Rightarrow m'$ e m'' são concorrentes.

Estas propriedades indicam que dado duas mensagens de difusão endereçadas ao mesmo grupo, se uma precede a outra, então a primeira tem número de bloco menor que a segunda, caso contrário, elas são concorrentes.

Um *Bloco Causal* é um vetor de tamanho $n = |g_x|$. p_i constrói um bloco causal para representar os eventos de envio e recebimento de mensagens com o mesmo número de bloco. A construção de tais vetores é estruturada na forma de uma matriz, onde as linhas representam uma fase ou bloco causal, e as colunas representam as difusões dos membros do grupo. Quando p_i difunde uma mensagem com número de bloco β ele marca a i-ésima entrada da linha β com um '+'; e para toda mensagem de difusão recebida com número de bloco β de um outro processo p_j, $j \neq i$, ele marca a j-ésima entrada do vetor para '+'. Esta matriz será representada como $BM_{x,i}$ para o processo p_i do grupo g_x. Assim, qualquer mensagem m enviada ou recebida no grupo gx em p_i, será representada na linha $BM_{x,i}[m.b]$.

Quando um Bloco Causal com um determinado número β está completo em p_i, então as mensagens deste bloco podem ser entregues a p_i em uma ordem fixa e pré-determinada. Assumindo-se que o canal é FIFO, um bloco causal $BM_{x,i}[\beta]$, $\beta \geq 1$, está completo para p_i se, para qualquer j, $1 \leq j \leq n$, $BM_{x,i}[\beta'][j]$ é um '+' para algum β', $\beta' \geq \beta$. Isto significa que p_i já enviou (se $i = j$) ou recebeu (se $i \neq j$) uma mensagem m tal que $m.b \geq \beta$, assim $BM_{x,i}[\beta]$ está completo.

A fim de garantir que um determinado bloco se torne completo (sincronização das fases), sempre que p_i cria um novo Bloco Causal, como resultado do recebimento de uma mensagem de difusão com número β, é estabelecido um período de tempo (*timeout*) para o bloco $BM_{x,i}[\beta]$, se p_i ainda não difundiu alguma mensagem com um timestamp maior que β. Se este tempo expirar (p_i não tem mensagens de difusão a enviar), p_i transmite para o grupo g_x uma mensagem nula, $timesilence_i$, com o maior timestamp que ele já recebeu e incrementa o seu contador para este novo valor:

AC3: (*Avança Contador devido a timesilence*$_i$): Antes de difundir uma mensagem nula m em um grupo g_x, p_i estabelece $m.b = max\{m'.b \mid receba_i(m') \wedge m'.g = g_x\}$, e
$BC_i = max\{BC_i, m.b\}$.

Como podemos observar, este tipo de mensagem nula tem o mesmo propósito das mensagens *resinc* do modelo de Raynal e Mostefaoui: manter sincronizados os processos quanto a criação de novas fases e a entrega das mensagens das fases anteriores.

O Newtop estabelece duas condições de segurança para a entrega das mensagens a um processo p_i:

s1: mensagens representadas em Blocos Causais com timestamp β são entregues em uma ordem fixa e pre-determinada depois que, para todo $g_x \in G_i$, $BM_{x,i}[\beta'], \beta' \geq \beta$, estão completos.

s2: mensagens representadas em $BM_{x,i}[\beta]$ são entregues somente depois que as mensagens em $BM_{x,i}[\beta_o]$ foram entregues, para todo $\beta_0 < \beta$.

Sempre que um processo difunde ou recebe uma mensagem não-nula, com número de bloco β, dentro do grupo x, ele difunde uma mensagem nula com timestamp β para todo grupo y cuja a matriz de blocos contém Blocos Causais com números menores que β. Isto fará com que cada membro do grupo y crie um Bloco Causal de número β, e assim todos os grupos estarão globalmente sincronizados com a ordem das mensagens.

O Algoritmo

Cada processo $p_i \in g_x$ mantém as seguintes variáveis:

- BC_i: Contador de blocos. Determina a ordem total. É iniciado com zeros.
- $BM_{x,i}$: Matriz de blocos para cada grupo $g_x \in G_i$. Cada linha desta matriz representa uma fase de difusão. É iniciada vazia.
- CBV_i: Vetor de blocos completos. Seu tamanho é $|G_i|$ (ou seja, o número de grupos aos quais p_i pertence). $CBV_i[x]$ indicará o maior número de blocos já completos em $BM_{x,i}$. É iniciado com $[0, 0, ..., 0]$.
- B_{min}: Menor elemento do vetor CBV_i. $B_{min} = min\{CBV_i[x] | g_x \in G_i\}$.

Sempre que o valor de B_{min} muda, com envios ou recebimentos, as mensagens não entregues com número de bloco menor ou igual a (o novo valor de) B_{min} são entregues de acordo com as condições de entrega descritas acima. Usaremos a notação $max_{x,i}$ para representar o maior número de bloco na matriz $BM_{x,i}$.

Os três procedimentos principais – envio, recebimento e entrega, são vistos no Algoritmo 4.4. No procedimento $envia(m)$, p_i difunde uma mensagem não-nula m para o grupo destino $m.g$, e difunde uma mensagem nula para aqueles grupos $g_x \in G_i$ onde $m.b > max_{x,i}$ (linhas 5 a 8).

A numeração dos blocos e a difusão de uma mensagem nula em resposta ao *timeout* expirado pelo mecanismo de *timesilence* procederão como visto em **AC3**. Assumimos que a difusão de uma mensagem nula ou não-nula automaticamente atualizará a correspondente matriz de bloco $BM_{x,i}$ e modificará CBV_i se necessário. Assumimos também, uma primitiva para entregar uma mensagem cuja execução modificará o contador local BC_i de acordo com **AC2** e entregará a mensagem à aplicação, se esta não for nula.

Quando uma mensagem m não-nula é recebida em $receba_i(m)$, uma mensagem nula é também difundida nos grupos $g_x \in G_i$, onde $m.b > max_{x,i}$ (linhas 11 a 21). O procedimento de entrega aguarda o valor de B_{min} mudar (ou incrementar, para ser mais preciso), e quando isto acontece, ele entrega todas as mensagens enviadas ou recebidas, que tenham número de bloco menor ou igual a B_{min}, em uma ordem pré-determinada e total (linhas 21 a 24).

Envio:

Evento $envia_i(m)$ **para o grupo** $m.g$

$m.b := BC_i + 1$; 1
$BC_i := m.b$; 2
divulgue m para os outros membros de $m.g$; 3
prepare uma mensagem nula n com $n.b := m.b$; 4
para cada g_x em $G_i - \{m.g\}$ faça 5
  se $m.b > max_{x,i}$ então 6
    $n.g := g_x$; 7
    divulgue n para os outros membros de g_x; 8
  fim_se
fim_para

Recebimento:

Evento $receba_i(m)$

atualize a matriz $BM_{m.g,i}$ correspondente; 9
atualize CBV_i; 10
se m é não-nula então 11
  $G' := G - \{m.g\}$; 12
  para cada g_x em $G - \{m.g\}$ faça 13
    se $m.b \leq max_{x,i}$ então 14
      $G' := G' - \{g_x\}$; 15
    fim_se
  fim_para
  se $G' \neq \{\}$ então 16
    prepare uma mensagem nula n com $n.b := m.b$; 17
    $BC_i := max\{n.b, BC_i\}$; 18
    para cada g_x em G' faça 19
      $n.g := g_x$; 20
      divulgue n para os outros membros de g_x; 21
    fim_para
  fim_se
fim_se

Entrega:

Para cada atualização em CBV_i

se $B_{min} \neq min\{CBV_i[x] \mid g_x \in G\}$ então 22
  $B_{min} := min\{CBV_i[x] \mid g_x \in G\}$; 23
  para toda mensagem m t.q. $m.b \leq B_{min}$ faça 24
    entregue m na ordem pré-determinada e total 25
fim_se

Algoritmo 4.4: Protocolo Newtop.

4.3.1 Corretude

Antes de apresentar os argumentos da corretude do protocolo, observe que no envio e recebimento de mensagens o número de bloco das mensagens nulas a serem difundidas é escolhido de forma que não viola a propriedade **pr1**, baseado no fato de que a completude dos blocos causais é identificada. Desta forma, os blocos causais completos são corretamente identificados durante a execução do protocolo.

Propriedade de segurança

Considere duas mensagens não-nulas m e m', tais que p_i e p_j estão em $m.g \cap m'.g$. Se $envia(m) \rightarrow envia(m')$ então $m.b < m'.b$. Como B_{min} nunca decrementa, não é possível $B_{min} \geq m'.b$ tornar-se verdadeiro antes que $B_{min} \geq m.b$ se torne. Desde que as mensagens são entregues em ordem crescente de seus números de bloco, $entrega(m) \rightarrow entrega(m')$ será verdadeiro tanto para p_i como para p_j. Suponha que $m.b = m'.b$. Para todo processo p_k em $m.g \cap m'.g$, quando $B_{min} \geq m.b$ torna-se verdadeiro pela primeira vez, $BM_{m.g,k}[m.b]$ e $BM_{m'.g,k}[m'.b]$ estarão completos; assim, m e m' serão entregues "juntas" e as suas ordens de entrega estarão de acordo com a ordem pre-determinada e fixada por todos os processos no sistema. Assim, o protocolo garante entrega em ordem total.

Propriedade de entrega

Considere uma mensagem não-nula m difundida por p_i em g_x. p_i assegura que uma mensagem nula n com $n.b = m.b$ é difundida de forma que para cada $g_y \in G_i - \{g_x\}$, um $BM_{y,i}[m.b]$ é criado, se ainda não o foi. Quando p_j, $p_j \in g_x$, recebe m, ele também assegura que $BM_{y,j}[m.b]$ é criado (se ainda não o foi) para todo $g_y \in G_j$. Cada bloco causal criado que está completo é identificado como tal, e assim, $B_{min} \geq m.b$ em um dado momento será verdadeiro para p_i e p_j. Portanto, m será finalmente entregue a p_i e p_j.

4.3.2 Desempenho

Dado que para uma simples difusão por um processo p_i em um grupo com n processos, o protocolo necessita que todos os processos p_j, $i \neq j$, enviem uma outra mensagem de difusão ou uma mensagem nula (caso não tenha difusões a fazer), então o desempenho geral do protocolo é n mensagens por difusão, em um único grupo de comunicação com n processos.

Se tomarmos a ordenação entre grupos, o desempenho passa para $n.|G_i|$, isto é, n vezes o número de grupos aos quais p_i pertence.

4.3.3 Exemplo

A Figura 4.5 mostra um exemplo de operação do Newtop, para dois grupos de processos, $G1 = \{p_1, p_2\}$ e $G2 = \{p_2, p_3\}$. Na fase 1, p_1 envia uma mensagem para p_2 no grupo $G1$, e p_2 envia uma mensagem para p_3 em $G2$. Em seguida, p_3 inicia a fase 2 enviando uma mensagem mensagem

para p_2. Como p_2 não recebeu nenhuma mensagem do grupo $G1$ desta fase, ele envia uma mensagem $nula(2)$ para $G1$ para manter sincronizados os grupos aos quais ele pertence. Em p_1, após a recepção da mensagem nula de p_2, ele cria um bloco causal para a fase 2 e inicia o temporizador. Como ele não tem nenhuma mensagem para enviar nesta fase, após o *timeout*, ele envia uma mensagem $timesilence(2)$ para p_2.

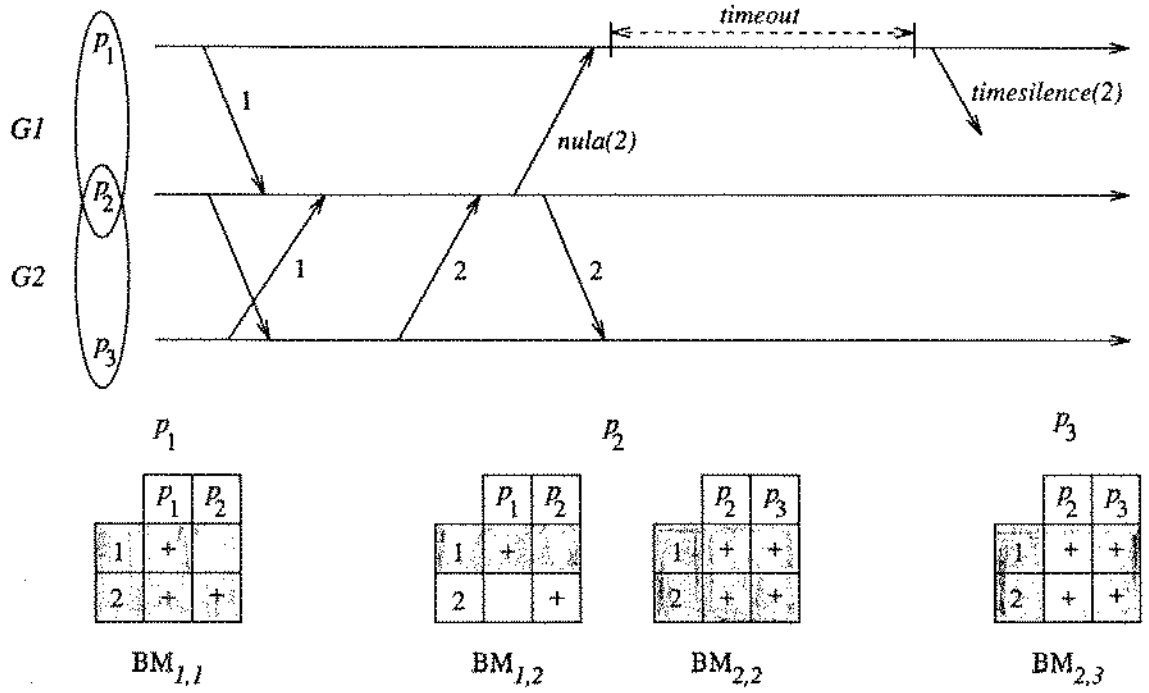


Figura 4.5: Preenchimento das Matrizes de Blocos.

O estado das matrizes de bloco dos processos, antes de $timesilence(2)$ chegar em p_2, são mostradas abaixo. A hachura em um bloco causal indica a completude do mesmo. $BM[1]_{1,1}$ torna-se completo no momento em que p_1 recebe a mensagem $nula(2)$ de p_2. Para completar $BM[2]_{1,1}$, p_1 envia outra mensagem nula ($timesilence(2)$) para p_2. Em p_2, somente o primeiro bloco causal está completo, pois o envio de uma mensagem nula na fase 2 completou este bloco. Até o momento em que p_2 recebe $timesilence(2)$ de p_1, o seu segundo bloco estará incompleto. No grupo $G2$, todos os blocos estão completos já que os processos trocaram mensagens em cada fase.

4.4 Comentários

No sistema ISIS, mudanças na configuração dos membros do grupo são gerenciadas através do conceito de sincronismo virtual, usando a primitiva GBCAST que fornece uma ordem total e global para todo o sistema; o mecanismo de ABCAST somente fornece ordenação de mensagens dentro de um grupo. Uma mensagem GBCAST causa uma parada temporária na ordenação de mensagens e uma descarga de todas as mensagens pendentes para assegurar a consistência

na ordenação global do sistema para as mensagens. Cada novo grupo é referenciado como uma visão, e a visão atual é adicionada ao vetor de timestamps de cada mensagem. O protocolo de gerenciamento do grupo do sistema ISIS é descrito em [RB91].

O sistema de programação distribuída Horus [BM96, HK95], que surgiu a partir do sistema ISIS, também implementa serviços de comunicação em grupo, e fornece difusão confiável causal e total. O Horus é todo dividido em camadas e altamente configurável, permitindo que as aplicações usem apenas os serviços que elas necessitam. O conjunto de camadas incluem a camada COM que fornece difusão não-confiável básica, a camada NAK que fornece difusão confiável FIFO, a camada MBRSHIP que fornece gerenciamento dos membros do grupo, a camada STABLE que possibilita estabilidade de mensagens, a camada FC que fornece controle de fluxo de mensagens, as camadas CAUSAL e TOTAL que, como o próprio nome diz, fornecem difusão causal e total, respectivamente; a camada LWG que mantém o grupo de processos e a camada EVS que mantém o sincronismo virtual estendido, entre outras. Técnicas avançadas de gerenciamento de memória são usados a fim de diminuir o custo deste mecanismo de divisão em camadas.

O Newtop substitui a idéia de grafo de contexto (discutida no próximo capítulo) do sistema Psync [LLPS89] pela noção de blocos causais. Cada bloco causal define um conjunto de mensagens. Todas as mensagens dentro de um bloco são causalmente independentes, e assim, os blocos são totalmente ordenados. Ou seja, as mensagens em um bloco são entregues no mesmo tempo lógico, em alguma ordem determinística. Desta forma, o Newtop fornece entrega totalmente ordenada similar à técnica de *ondas* do Psync e do mecanismo *todos-reconhecem* do Lansis [YAM92b], mas com menos "contabilidade". A entrega causal do Newtop é menos eficiente que a do Psync ou do Trans [PMMSA90], pois a informação causal representada nos blocos causais não é acurada e é mais pessimista que o necessário (embora, mais compacta). Além disso, usar blocos causais elimina a necessidade de usar algoritmos mais rápidos (como ToTo [Kra93]) que usam o grafo de contexto inteiro para conseguir decisões mais rápidas na ordem total. O Newtop implementa um serviço de gerenciamento do grupo que lida com falhas de processos e partições de rede. Entretanto, recuperação de processos e re-junção de rede não são tratadas.

Capítulo 5

Protocolos baseados em Grafo de Contexto

Neste capítulo descrevemos os protocolos que se utilizam de *Grafo de Contexto* como principal estrutura de controle para determinar a ordem das mensagens de difusão. Para isto, cada processo envia uma mensagem no contexto daquelas mensagens que já foram enviadas ou recebidas. Informalmente, "no contexto de" define uma relação que é representada na forma de um grafo direcionado acíclico, chamado de *Grafo de Contexto*, e que denota a relação de causalidade entre as mensagens.

A partir da ordem parcial determinada pelo grafo de contexto, os processos podem chegar a uma ordem total das mensagens, através da determinação das mensagens concorrentes causalmente. Estas mensagens concorrentes no grafo de contexto podem ser entregues em alguma ordem determinística.

5.1 Psync

O protocolo *Psync* (de "*pseudosynchronous*"), desenvolvido por Peterson, Buchholz e Chlichting [LLPS89], foi o primeiro a introduzir o conceito de grafo de contexto, e a usar as informações de causalidade nas próprias mensagens trocadas entre os processos pertencentes a um grupo de comunicação.

O Psync não implementa ordenação total, mas a ordenação causal que ele consegue determinar pode ser usada para se chegar a uma ordenação total das mensagens, caso todos os membros do grupo produzam a mesma ordem topológica do grafo de contexto. Esta idéia serviu como base para os dois protocolos seguintes descritos neste capítulo, o Trans/Total e o Transis.

Descrição do Protocolo

Seja P o conjunto dos participantes de uma conversação (grupo de comunicação), e M o conjunto de mensagens trocadas no grupo. Defina "$\rightarrow$" como sendo uma relação transitiva em M, tal que $m \rightarrow m'$ se e somente se m é enviada no contexto de m'; ou seja, o processo que enviou

m' já enviou ou recebeu m. Seja $G_{\rightarrow}$ o grafo direcionado acíclico que representa as relações de contexto, e G a redução transitiva de $G_{\rightarrow}$. Isto é, G contém todos os vértices e nenhuma aresta redundante de $G_{\rightarrow}$, onde uma aresta (m, m') é dita redundante se $G_{\rightarrow}$ também contém um caminho de m a m' de comprimento maior que um [GU72].

A Figura 5.1 mostra um exemplo de $G_{\rightarrow}$ e G para uma conversação entre quatro processos, p_1, p_2, p_3 e p_4. A mensagem m_1 é a mensagem inicial enviada por p_1; m_2 e m_3 foram enviadas por processos que receberem m_1, mas independentes um do outro, criando duas seqüências lógicas de mensagens (linhas pontilhadas na figura); m_4 foi enviada por um processo que recebeu m_3, mas não recebeu m_2; e m_5 foi enviada no contexto de todas as outras mensagens. Os nós aos quais uma determinada mensagem está ligada em G são ditos *predecessores* imediatos da mensagem. Por exemplo, m_2 e m_4 são os predecessores imediatos de m_5. Quando duas mensagens não estão no contexto de uma para a outra, elas são ditas como estando no *mesmo tempo lógico*. Por exemplo, m_2 e m_3 estão no mesmo tempo lógico.

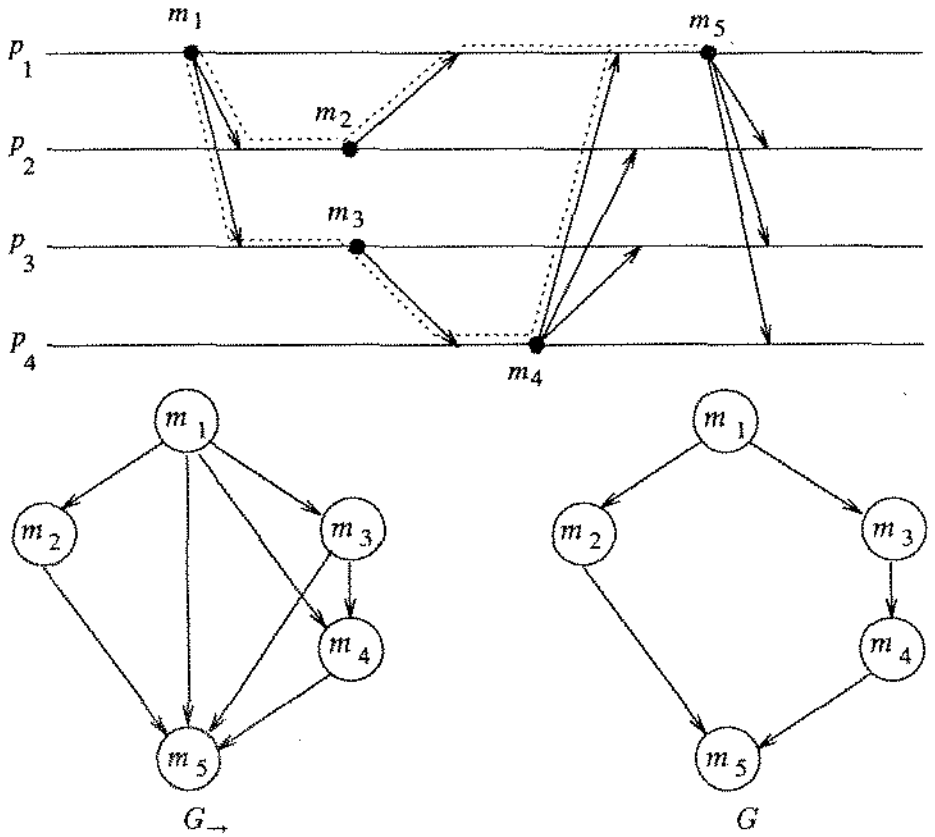


Figura 5.1: Exemplo de Grafo de Contexto.

Cada participante em uma conversação tem uma *visão* do grafo de contexto que corresponde àquelas mensagens que ele enviou ou recebeu. Seja $M_p \subseteq M$ o subconjunto de mensagens enviadas ou recebidas pelo membro (participante) $p \in P$. A visão do processo p, V_p, é uma restrição de G para os vértices em M_p e as arestas em E incidentes sobre estes vértices. Um

processo com uma visão igual a G recebeu todas as mensagens enviadas pelos outros membros. Mensagens fora da visão do membro são ditas *em-espera*. Por exemplo, no momento em que um participante enviou m_4, sua visão consistia de m_1 e m_3; m_2 estava *em-espera*.

A operação *receive* retorna uma mensagem *em-espera* m em G tal que não há uma outra mensagem *em-espera* m' para a qual $m' \rightarrow m$. Esta mensagem m é adicionada à visão de p, V_p. Para qualquer par de mensagens m e m' recebidas por um processo, m é recebida antes de m' se $m \rightarrow m'$. Assim, quando um processo recebe uma determinada mensagem, é garantido que ele já recebeu todas as mensagens que a precederam no grafo de contexto.

Quando um processo p aplica a operação *send* para uma mensagem m, m é adicionada a M e a aresta (m_l, m) é adicionada a E para cada nó m_l que é uma folha de V_p. As informações das arestas adicionadas a G são passadas na mensagem. Note que as estruturas de dados que representam uma conversação incluem um grafo de contexto "compartilhado" e uma visão "privada" de cada participante como uma janela deste grafo G.

Note, também, que as operações *send* e *receive* modificam o grafo de contexto de tal modo que G permanece a redução transitiva de $G_{\_}$.

O grafo de contexto contém informações sobre quais processos receberam quais mensagens. Em particular, a recepção de uma mensagem implica que o transmissor conhece todas as mensagens predecessoras. Formalmente, uma mensagem m_p enviada por um processo p torna-se *estável* se para cada participante $q \neq p$, existe um vértice m_q em G enviado por q, tal que $m_p \rightarrow m_q$. Intuitivamente, cada mensagem m_q serve como um reconhecimento de m_p de algum processo q. Para uma mensagem tornar-se estável é necessário que todos os processos a tenham recebido, e assim, todas as futuras mensagens enviadas para o grupo devam estar no contexto daquelas mensagens estáveis; isto é, elas não podem preceder ou estar no mesmo tempo lógico das mensagens estáveis.

A ordenação topológica do grafo de contexto, construída por todos os participantes para se conseguir uma ordem total das mensagens, deve ser incremental de modo que cada processo espera por uma porção de sua visão estabilizar antes de permitir que a ordenação seja aplicada. Isto é para assegurar que nenhuma mensagem futura enviada para o grupo invalidará a ordem total.

Considere o esquema da Figura 5.2. Cada visão dos participantes é conceitualmente particionada em dois subgrafos, *entregues* e *não-entregues*, denotados por V_p^c e V_p^u, respectivamente. As linhas pontilhadas mostram um possível caminho entre duas mensagens (nós). O subgrafo V_p^c corresponde àquelas mensagens que já foram totalmente ordenadas e entregues à aplicação. O subgrafo V_p^u corresponde ao conjunto de mensagens que ainda serão ordenadas. Cada interação da ordenação topológica incremental move-se através de V_p em *ondas*, onde uma onda é o conjunto máximo de mensagens enviadas no mesmo tempo lógico; isto é, a relação de contexto não se aplica entre qualquer par de mensagens de uma onda. Tão logo a onda esteja *completa* – isto é, o participante está certo de que nenhuma mensagem futura pertencerá a esta onda – as mensagens na onda são ordenadas de acordo com algum algoritmo de ordenação determinístico e entregues à aplicação. Estas mensagens são, então, movidas de V_p^u para V_p^c.

Um importante problema que surge é determinar quando todas as possíveis raízes de V_p^u

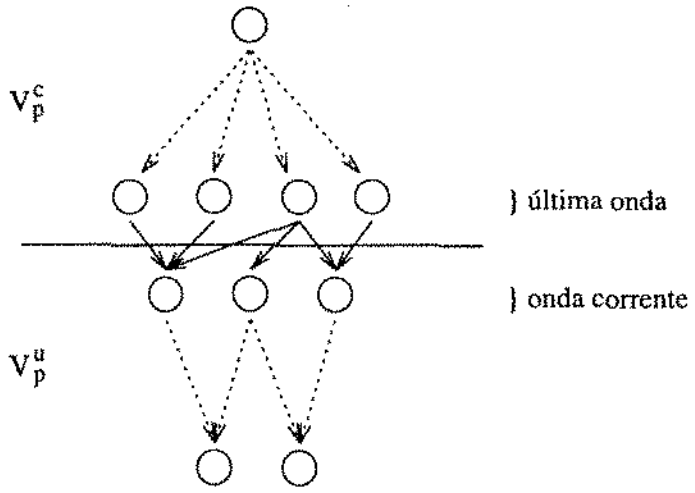


Figura 5.2: Partições de V_p.

estão presentes. Quando uma mensagem torna-se estável, todas as futuras mensagens devem segui-la no grafo de contexto. Assim, uma simples mensagem estável em uma determinada onda implica que todos os membros possíveis da onda estão contidos na visão de um participante. Em outras palavras, tão logo uma raiz de V_p^u torne-se estável, todas as raízes de V_p^u podem ser ordenadas e entregues à aplicação. Em contraste, considere as condições mais fraca e mais forte para entrega de uma mensagem. Por um lado, não é correto entregar uma mensagem tão logo ela se torne estável. Isto porque a ordem na qual as mensagens tornam-se estáveis em duas visões diferentes pode diferir devido às variações de atraso no meio de comunicação, resultando em ordenações totais potencialmente diferentes. Por outro lado, não é necessário esperar que todas as mensagens na onda tornem-se estáveis antes de entregar tal onda; uma simples mensagem estável na onda já é suficiente. Assim, uma onda pode ser entregue se ela é a onda raiz de V_p^u e uma das suas mensagens tornou-se estável.

Estruturas de Dados

Cada processo mantém uma fila, Q_s, contendo as mensagens geradas pela aplicação e que aguardam ser enviadas. As mensagens de difusão têm um cabeçalho com as seguintes informações:

- $AN(AddNode)$: informa que é uma mensagem de difusão;
- *cid*: identificador da conversação (grupo de comunicação);
- *mid*: identificador da mensagem;
- pid_{sender}: identificador do processo transmissor;
- $pred\_mid_1, ..., pred\_mid_n$: identificação das mensagens predecessoras imediatas da mensagem no grafo de contexto;

Seja I_p a imagem local que o processo p tem do grafo de contexto G. Cada mensagem AN que chega ao processo p é imediatamente inserida em I_p, se todas as predecessoras a ela estão presentes em I_p. Se uma ou mais predecessoras da mensagem ainda não chegaram em p, então a mensagem é posta em uma fila de espera até que todas as predecessoras estejam presentes.

Seja m uma mensagem enviada por um participante p no contexto de m', e seja p' o participante que recebeu m mas não recebeu m'; isto é, m é posta na fila de espera de p'. O protocolo associa um temporizador para cada mensagem da fila de espera. Quando o tempo para a mensagem m expira, um pedido de retransmissão de m' é enviado para p.

Uma mensagem de pedido de retransmissão, identificada como RR, informa o subgrafo de G que necessita ser retransmitido, pois é possível que as predecessoras das predecessoras também estejam faltando. Este tipo de mensagem tem as seguintes informações, além do campo *cid* descrito anteriormente:

- *mid*: identifica a mensagem cujo as predecessoras estão faltando;
- $leaf\_mid_1, ..., leaf\_mid_n$: identifica as folhas da imagem local que estão faltando.

Toda vez que uma mensagem é recebida por um processo, se este processo não tem mensagens a envia, então ele envia uma mensagem de reconhecimento, ACK, para os outros processos, contendo os identificadores das mensagens que chegaram.

O Algoritmo

O Algoritmo 5.1 mostra o procedimento que cada participante executa. A operação *wait_input*() é usada para bloquear o processo a fim de esperar pelas mensagens das múltiplas fontes.

O núcleo do procedimento de difusão são os dois conjuntos de nós *last_wave* e *current_wave*, correspondendo as folhas de V_p^c e as raízes de V_p^u, respectivamente. Quando iniciado, o procedimento chama a rotina *initialize*() que inicia um grupo de comunicação e retorna a última onda do Grafo de Contexto deste grupo. Em seguida vem o ciclo principal do algoritmo. Primeiro ele adiciona todos os dependentes conhecidos dos nós em *last_wave* para *current_wave* (linhas 5 e 6 do algoritmo). Depois, ele checa, através da função *stable*(), se algum dos nós em *current_wave* está estável. Caso positivo, *current_wave* é atribuído a *last_wave*, e a rotina *sort*() é chamada para ordenar as mensagens em *current_wave*. Por fim, as mensagens ordenadas são entregues à aplicação (linhas 7 a 11). Assuma que a rotina *sort*() remove qualquer mensagem em *current_wave* que não seja para a aplicação, como exemplo, mensagens de reconhecimento ACK. Esta mesma rotina deve ser aplicada por todos os participantes do grupo, pois ela determina a ordem total para a entrega das mensagens (linha 11). Se nenhuma das mensagens está estável, então o algoritmo espera que novas mensagens cheguem (linha 12) e verifica a estabilidade de *current_wave* na próxima execução do laço principal. O laço nas linhas 13 a 15 faz com que o processo espere por alguma mensagem pendente, usando a função *em-espera()* que retorne verdadeiro caso haja tal mensagem. Assuma que existe um mecanismo de pedido de retransmissão quando uma mensagem pendente não chega após um período de

```
Difusão:
    Quando participar de uma conversação conv:
        conv, last_wave := initialize();                                          1
        enquanto (VERDADEIRO) faça
            enviou_algo := FALSO;                                                 2
            recebeu_algo := FALSO;                                                3
            current_wave := φ;                                                    4
            para cada nó ∈ last_wave faça                                         5
                current_wave := current_wave ∪ próximo(nó);                       6
            fim_para
            se (∃nó ∈ current_wave, tal que stable(nó, conv)) então               7
                last_wave := current_wave;                                        8
                sort(current_wave);                                               9
                para cada nó ∈ current_wave faça                                  10
                    entregue nó à aplicação;                                      11
                fim_para
            fim_se
            wait_input();                                                         12
            enquanto (em_espera(conv)) faça                                       13
                nó, msg := receive(conv);                                         14
                recebeu_algo := VERDADEIRO;                                       15
            fim_enquanto
            enquanto (não vazia Q_s) faça                                         16
                msg := próximo(Q_s);                                              17
                send(msg, conv);                                                  18
                enviou_algo := VERDADEIRO;                                        19
            fim_enquanto
            se ((não enviou_algo) ∧ recebeu_algo)) então                          20
                send(ACK, conv);                                                  21
            fim_se
        fim_enquanto
```

Algoritmo 5.1: Protocolo Psync.

tempo (*timesout*). No laço seguinte (linhas 16 a 19) as mensagens geradas pelo processo são enviadas. Note que as mensagens em Q_s são enviadas depois de receber todas as mensagens *em-espera* do grupo. Isto faz com que toda mensagem nova enviada reconheça as mensagens recebidas. Finalmente, uma mensagem de reconhecimento explícito, ACK, é enviada somente se alguma mensagem é recebida e nenhuma é enviada (linhas 20 e 21). Este ACK reconhece o recebimento das mensagens que chegaram.

5.1.1 Desempenho

O custo na entrega em ordem total das mensagens de difusão no protocolo Psync é no mínimo n, onde n é o número de processos do grupo de comunicação. Isto porque as mensagens de uma onda só são entregues quando estas tornam-se estáveis, e para isto todos os processos devem difundir alguma mensagem que segue na ordem parcial qualquer mensagem da onda.

Para entrega em ordem causal o custo é 1, ou seja, é o custo da própria difusão, contanto que o processo já tenha recebido todas as mensagens que precedem esta difusão.

5.1.2 Exemplo

A Figura 5.3 mostra uma possível visão do Grafo de Contexto do Psync para um grupo de comunicação com seis processos. Os processos A e D geram duas mensagens inicialmente, A_1 e D_1. Em seguida, B gera a mensagem B_1 no contexto da mensagem A_1, e C gera a mensagem C_1 no contexto de D_1. A mensagem A_2 reconhece o recebimento das quatro mensagens iniciais por A, assim como E reconhece através de E_1 o recebimento de C_1 e D_1. A mensagem F_1 por F reconhece todas as mensagens enviadas no grupo para este exemplo. A partir deste grafo, podemos deduzir as ondas para entrega das mensagens: $\{A_1, D_1\}$, $\{B_1, C_1\}$, $\{A_2, E_1\}$, $\{C_2\}$ e $\{F_1\}$.

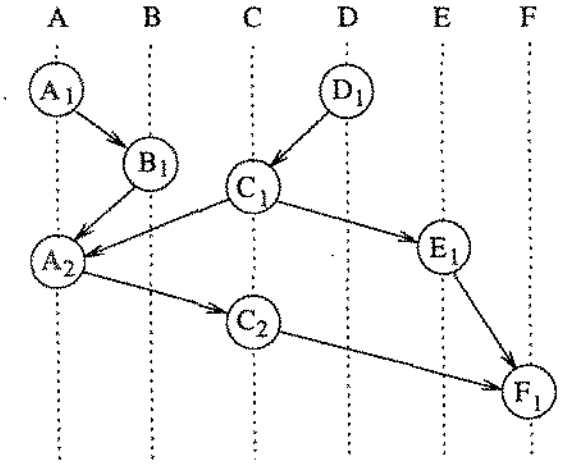


Figura 5.3: Exemplo de operação do Psync.

Observe que as mensagens de uma onda se tornam estáveis apenas quando todos os processos

do grupo difundem alguma informação a respeito do recebimento destas mensagens. No exemplo da Figura 5.3, no momento em que o processo F difunde F_1, todos os processos podem decidir que as mensagens $\{A_1, D_1\}$ estão estáveis. Desse modo, se algum dos processos do grupo falhar, todo o sistema ficará travado até que esse processo seja removido do grupo, visto que os outros processos precisam saber do recebimento das mensagens por tal processo.

5.2 Trans/Total

O protocolo *Trans/Total*, desenvolvido por Melliar-Smith, Moser e Agrawala [PMMSA90], tem uma abordagem semelhante à do protocolo *Psync*. Ou seja, constrói uma ordem parcial das mensagens de difusão que pode ser convertida em uma ordem total. Entretanto, diferentemente do Psync, o Trans/Total não requer que o sistema seja parcialmente síncrono e fique bloqueado até que um membro que falhou seja detectado e removido da configuração.

A semelhança dos dois algoritmos se dá através da utilização, também no Trans, do conceito de grafo de contexto. A mesma idéia é implementada através do modelo OPD (*Observable Predicate for Delivery*), que além de representar a ordem parcial das mensagens trocadas entre os processos, determina, também, as mensagens não recebidas por cada processo.

5.2.1 O Protocolo Trans

O protocolo de difusão Trans usa uma combinação de reconhecimentos positivo e negativo para alcançar uma comunicação confiável em um meio onde falhas de transmissão podem ocorrer.

Assim como no Psync, a idéia principal do Trans é que um reconhecimento para as difusões são repassados nas próprias mensagens de difusão e tipicamente será visto por todos os membros do grupo. Cada mensagem de difusão carrega o identificador do remetente e um número de seqüência para a mensagem, gerado a partir de um contador de difusões do remetente.

A idéia por trás do protocolo é ilustrada por esta seqüência de eventos em um sistema consistindo de três processos: P, Q e R [PMMSA90]:

- O processo P difunde uma mensagem com um número de seqüência igual ao valor de seu contador;
- A mensagem de P é recebida pelo processo Q;
- O processo Q inclui um reconhecimento positivo para a mensagem de P em sua próxima mensagem de difusão;
- O processo R ao receber a difusão de Q está ciente de que a mensagem de P já foi reconhecida e por isso R não necessita reconhecê-la na sua próxima difusão; ao invés disso, R reconhece a mensagem de Q;
- Se o processo R não recebeu a mensagem de P, então a mensagem de Q alerta R desta perda de mensagem, e assim, R inclui um reconhecimento negativo para a mensagem de P em sua próxima difusão.

O melhoramento proposto pelo Trans se dá pelo fato de que não é necessário reconhecer positivamente todas as mensagens já difundidas. Além disso, as mensagens carregam os reconhecimentos negativos das mensagens ainda não recebidas.

Ainda, as retransmissões das mensagens podem ser feitas por qualquer processo membro, não apenas pelos processos que as originaram. A mensagem retransmitida contém os mesmos reconhecimentos, positivo e negativo, da mensagem original.

Para evitar grandes atrasos em um sistema com poucas difusões, se um processo não tem mensagens a transmitir, ele constrói uma mensagem nula que difunde os seus reconhecimentos. O atraso aceitável antes de transmitir uma mensagem nula pode diferir para reconhecimentos positivos e negativos.

Estruturas de Dados

Cada processo mantém as seguintes listas:

- *Lista de Reconhecimentos* (LR): contém os identificadores das mensagens para as quais o processo tem que enviar um reconhecimento;
- *Lista de Reconhecimentos Negativos* (LRN): contém os identificadores das mensagens que o processo ainda não recebeu.
- *Lista de Mensagens Recebidas* (LMR): contém as mensagens já recebidas e que podem necessitar de retransmissões. Mensagens são removidas desta lista quando retransmissões destas mensagens não são mais necessárias para qualquer processo.
- *Lista de Retransmissões Pendentes* (LRP): contém o identificador das mensagens cuja retransmissão foi requisitada por algum processo, isto é, o processo recebeu a mensagem e um reconhecimento negativo para cada mensagem desta lista.

Cada mensagem m contém os seguintes campos:

- $m.id$: identificador da mensagem, contendo a identificação do processo remetente e o seu número de seqüência;
- $m.acks$: lista de identificadores de mensagens reconhecidas positivamente através de m. Isto é, o processo que enviou m reconhece que recebeu todas as mensagens em $m.acks$;
- $m.nacks$: lista de identificadores de mensagens reconhecidas negativamente através de m. Isto é, o processo que enviou m informa que não recebeu as mensagens em $m.acks$;

Os passos para um processo difundir e receber uma difusão é mostrado no Algoritmo 5.2. Ao difundir uma mensagem, o processo retira de LR os reconhecimentos positivos inseridos na mensagem, mas retém em LRN os reconhecimentos negativos (linhas 1 a 4). Se há muitos reconhecimentos a serem transmitidos, então os reconhecimentos negativos têm prioridade sobre

Difusão:

Ao enviar uma mensagem m**:**

$\forall id \in$ LR insira em $m.acks$;	1
remova $\forall id \in$ LR;	2
$\forall id \in$ LRN insira em $m.nacks$;	3
difunda m;	4

Ao receber uma mensagem m**:**

adicione $m.id$ a LR;	5
adicione m a LMR;	6
se ($m.id \in$ LRN) então	
remova $m.id$ de LRN;	7
se ($m.id \in$ LRP) então	
remova $m.id$ de LRP;	8
para todo id, tal que $id \in m.acks$ faça	9
se ($id \in$ LR) então	
remova id de LR;	10
se (mensagem correspondente a $id \notin$ LMR) então	
adicione msg correspondente a id a LRN;	11
fim_para	
para todo id, tal que $id \in m.nacks$ faça	12
se (mensagem correspondente a $id \in$ LMR) então	
adicione msg correspondente a id a LRP;	13
senão	
adicione id a LRN;	14
fim_para;	

Algoritmo 5.2: Protocolo Trans.

os positivos. Além disso, se um processo não recebe nenhum reconhecimento positivo dentro de um certo intervalo de tempo, então ele adiciona a mensagem à LRP (e depois a difunde novamente).

Quando um processo recebe uma mensagem m,

- Ele adiciona o $m.id$ em LR e armazena m na LMR (linhas 5 e 6). Se $m.id$ está em LRN, então este identificador é removido desta lista (linha 7), pois, assim, a mensagem foi recebida e não há mais necessidade de enviar um reconhecimento negativo da mesma. Similarmente, se a mensagem está presente na LRP, então o seu identificador também é removido desta lista (linha 8), de forma que a chegada da mensagem significa que algum outro processo satisfez o pedido de retransmissão e, portanto, ele não precisa mais re-enviar a mensagem.

- Todas as mensagens cujos reconhecimentos estão em m não precisam mais ser reconhecidas por este processo, e assim seus identificadores são removidas de LR (linhas 9 e 10). Se uma mensagem que é reconhecida em m está na LR, então seu identificador é removido desta lista, pois nenhum reconhecimento é necessário mais para esta mensagem. Se uma mensagem que é reconhecida em m não está na LR, então o processo deixou de receber esta mensagem e seu identificador é adicionado a LRN (linha 11).

- Se uma mensagem é reconhecida negativamente em m (linha 12), e se esta mensagem já foi recebida pelo processo, então ele a adiciona à LRP (linha 13), pois o remetente de m requisitou retransmissão. Se o processo ainda não recebeu esta mensagem, então ele adiciona seu id à LRN (linha 14).

- O processo pode também determinar que ele não recebeu uma mensagem de algum processo observando que o número de seqüência da mensagem é maior que o último número de seqüência mais um deste processo. Novamente, um ou mais reconhecimentos negativos são adicionados à LR.

Uma mensagem de difusão, juntamente com seus reconhecimentos, é retida na LMR até que o processo determine que todos os processos na configuração receberam a mensagem.

Exemplo

A Figura 5.4 mostra um exemplo da operação do Trans, com o grafo que representa os reconhecimentos positivos (linhas cheias) e negativos (linhas pontilhadas) das mensagens trocadas por quatro processos, A, B, C e D, dentro de um grupo. As setas significam o seguinte: se uma mensagem m_2 reconhece uma mensagem m_1 então há uma seta no grafo de m_2 para m_1. Observe que, com esta notação, o sentido das setas é invertido com relação ao modelo de grafo de contexto introduzido pelo Psync, e não representam diretamente a relação "$\rightarrow$".

Neste exemplo, foram geradas as seguintes mensagens de difusão: B_1, D_1, A_1d_1, $C_1\overline{d}_1b_1a_1$, $D_2\overline{a}_1c_1$, D_1, $C_2d_2d_1$, $B_2\overline{a}_1c_2$, onde a letra maiúscula identifica o processo que originou a mensa-

gem, e as letras minúsculas são os reconhecimentos informados na mensagem. Os identificadores com uma barra indicam um reconhecimento negativo da mensagem.

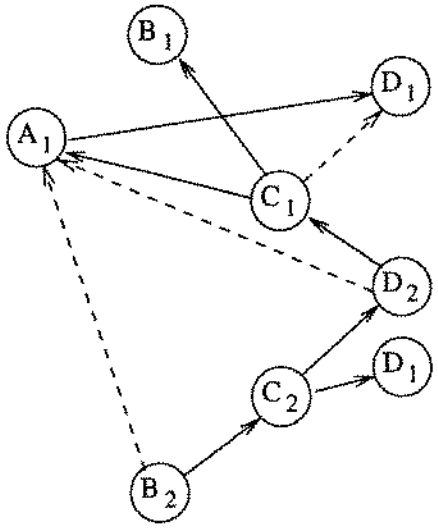


Figura 5.4: Reconhecimentos positivos e negativos das difusões.

Inicialmente, as mensagens B_1 e D_1 foram enviadas. A envia $A_1 d_1$ reconhecendo o recebimento de D_1. C informa em C_1 que não recebeu D_1 e que recebeu B_1 e A_1. D informa que não tem A_1 e que tem C_1, e assim ele também tem B_1, A_1 e D_1. Como D_1 foi requisitada por C, esta é enviada novamente. Ao receber D_1, C confirma este recebimento e o de D_2. Por fim, B informa que não recebeu A_1 mas que recebeu todas as outras, através da transitividade de C_2. Note que a mensagem D_2 é considerada a implicitamente reconhecer a mensagem D_1, pois ela foi enviada pelo mesmo processo.

5.2.2 O Protocolo Total

Até agora, nós discutimos o protocolo Trans, que assegura que uma mensagem difundida é recebida por todos os processos operacionais, mas não garante que mensagens diferentes sejam recebidas por processos diferentes na mesma ordem. Nesta seção, apresentamos o protocolo Total que estende o protocolo Trans para satisfazer as propriedades de ordenação total.

No protocolo Trans, visto que os reconhecimentos positivos e negativos são adicionados às mensagens difundidas, um processo pode determinar se um outro recebeu ou não uma mensagem. Para isto é aplicada a regra *Predicado Observável para Entrega* (*Observable Predicate for Delivery*), denotada por $OPD(P, A, C)$, onde P é um processo e A e C são mensagens. Denotaremos o remetente de A como P_A. Se $OPD(P, A, C)$ é verdadeiro, então ele declara que o processo P está certo que P_C recebeu e reconheceu, direta ou indiretamente, a mensagem A no momento da difusão de C. O processo P pode avaliar o predicado baseado nas mensagens que ele recebeu. Este predicado é verdadeiro, se e somente se, da seqüência de todas as mensagens recebidas, P pode formar uma seqüência S_M de mensagens tais que [PMMSA90]:

1. A seqüência começa com a mensagem A e termina com a mensagem C.

2. Cada mensagem de S_M, que não seja A, reconhece positivamente sua predecessora na seqüência, ou é uma difusão pelo processo que difundiu sua predecessora na seqüência.

3. Nenhuma mensagem em S_M é reconhecida negativamente pela mensagem C.

Essencialmente estas propriedades informam que P recebeu uma seqüência de mensagens (não necessariamente consecutivas) na qual os reconhecimentos, iniciando dos *acks* em C, transitivamente reconhecem A [Jal94].

Tomaremos como exemplo, o grafo da Figura 5.4. Este grafo representa a seqüência global de mensagens transmitidas. Em um dado momento, o grafo em um processo dependerá da seqüência de mensagens recebida por este processo. Entretanto, já que o Trans assegura a entrega das mensagens, e desde que uma retransmissão é exatamente igual à mensagem original, certamente em algum momento todos os processos terão um grafo que é igual ao grafo global. Se um processo recebe uma mensagem m_1 antes de transmitir uma mensagem m_2, então deve haver um caminho no grafo de m_2 para m_1. Por exemplo, na Figura 5.4, há um caminho de B_2 a C_1, e uma aresta de B_2 a C_2, implicando que B_2 reconhece C_2. Mas C_2 contém reconhecimentos para D_2 e D_1. Assim, no momento da difusão de B_2, o processo P_B deve ter recebido estas mensagens, senão ele teria que incluir um reconhecimento negativo para elas. Novamente, D_2 contém um reconhecimento negativo par A_1 e um positivo para C_1. Visto que B_2 não contém nenhum reconhecimento negativo para C_1, P_B deve ter recebido-a no momento de enviar B_2. Já que ele não recebeu A_1, então um reconhecimento negativo é adicionado a B_2.

$OPD(P, A, C)$ significa que há um caminho de C até A no grafo formado pelas mensagens recebidas por P e que não há uma aresta de reconhecimento negativo de C para qualquer mensagem no caminho de C até A. Isto é, C transitivamente reconhece A.

A regra OPD pode ser usada para determinar que uma mensagem foi recebida por todos os processos do grupo e, assim, poderá ser removida da LMR. Esta regra permite que os processos construam uma ordem parcial das mensagens de difusão:

Ordem Parcial: Na ordem parcial construída pelo processo P, uma mensagem C *segue* uma mensagem B se, e somente se, $OPD(P, B, C)$ e para toda mensagem A, $OPD(P, A, B)$ implica $OPD(P, A, C)$.

Na ordem parcial, se C segue A, então implica que C reconhece (direta ou indiretamente) a mensagem A e também todas as mensagens que A reconhece. Se C é incluída na ordem parcial, significa que no momento da transmissão de C, o processo P_C recebeu e reconheceu, direta ou indiretamente, todas as mensagens que precedem C na ordem parcial. A Figura 5.5 mostra a ordem parcial construída a partir dos reconhecimentos gerados no exemplo da Figura 5.4.

Note que nesta ordem parcial, a mensagem C_1 não segue A_1 (embora ela contenha um reconhecimento a_1) porque A_1 segue D_1, mas C_1 tem um reconhecimento negativo para D_1. Similarmente, B_2 não segue C_2 por causa de A_1. Entretanto, C_2 segue A_1 já que existe uma mensagem de C que reconhece $A_1 - C_1$.

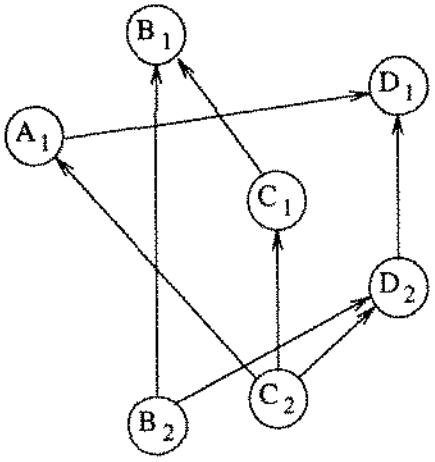


Figura 5.5: Ordem parcial derivada dos reconhecimentos das mensagens.

O objetivo do protocolo Total é conseguir uma ordem total das mensagens e assegurar que todos os processos ativos no sistema determinam esta mesma ordem. O Total é baseado na relação de ordem parcial derivada dos reconhecimentos das mensagens construídas pelo protocolo Trans. Há apenas uma ordem parcial, que deve ser a mesma para todos os processos, mas alguns processos podem estar cientes de apenas parte desta ordem parcial, já que eles podem não ter recebido todas as mensagens difundidas.

O Total não necessita de nenhuma mensagem adicional além daquelas requeridas pelo Trans. Entretanto, uma mensagem não é posta em ordem total imediatamente depois que ela é recebida. Um processo deve esperar receber mais mensagens dos outros processos antes de poder adicionar a mensagem à ordem total. Isto é, o protocolo incrementalmente estende a ordem total. O protocolo Total é também resistente a falhas dos processos.

Uma mensagem que segue na ordem parcial apenas aquelas mensagens já na ordem total (ou segue nenhuma outra mensagem) é uma mensagem *candidata* para inclusão na ordem total. Cada conjunto de mensagens candidatas, chamado de *conjunto candidato*, é votado pelas mensagens que seguem as candidatas na ordem parcial. Esta "votação" não significa efetivamente uma eleição envolvendo votos e troca de mensagens, mas é uma avaliação baseada nas mensagens recebidas. Desta forma, a decisão de um processo a cerca de incluir uma mensagem na ordem total é dependente apenas da seqüência de mensagens que ele recebeu.

Cada processo define o seu conjunto candidato, e cada conjunto é votado em separado. Esta idéia é equivalente ao modelo de ondas do protocolo Psync, onde o conjunto de mensagens de uma onda poder ser comparado ao conjunto de mensagens candidatas.

O processo de votação em um conjunto candidato (**CS**) necessita de uma série de estágios. O número de estágios de votação depende do conjunto candidato e da ordem parcial. Uma mensagem vota em um estágio somente se nenhuma mensagem anterior de seu remetente já votou neste estágio. No estágio 0, o voto de uma mensagem m em um conjunto candidato depende se esta mensagem segue na ordem parcial as mensagens deste conjunto. No estágio i, onde $i > 0$, uma mensagem vota em um conjunto candidato se ela segue na ordem parcial

mensagens suficientes que votaram no estágio $i-1$.

O número de votos requeridos para uma decisão e para um futuro voto deve ser pelo menos N_d e N_v, respectivamente. A tabela 5.1 mostra os valores de N_d e N_v para cada nível de resiliência. O algoritmo é resistente a um número de falhas menor que $\frac{n}{3}$, onde n é o número de processos no sistema.

Resiliência	N_d	N_v
1	$\frac{n+2}{2}$	$\frac{n-1}{2}$
2	$\frac{n+3}{2}$	$\frac{n-2}{2}$
$k < \frac{n}{3}$	$\frac{n+k+1}{2}$	$\frac{n-k}{2}$

Tabela 5.1: Número de votos requeridos por grau de resiliência do sistema.

Os critérios de votação

Uma mensagem m vota em um conjunto candidato CS como segue:

No estágio 0:

- m *vota em* CS se CS contém apenas m.
- m *vota em* CS, se (1) nenhuma mensagem do remetente de m que precede m votou em CS, (2) m segue toda mensagem em CS, e (3) m segue nenhuma outra mensagem candidata.
- m *vota contra* CS se ela segue alguma outra mensagem candidata que não esteja neste conjunto.

No estágio i, onde $i > 0$:

- m *vota em* CS se
  1. nenhuma mensagem do remetente de m que precede m votou em CS;
  2. o número de mensagens que votou em CS no estágio $i-1$ que segue m na ordem parcial é pelo menos N_v; e
  3. ela segue menos mensagens que votaram contra CS do que as que votaram a favor no estágio $i-1$.
- m *vota contra* CS se
  1. o número de mensagens que votaram contra CS, no estágio $i-1$ e que m as segue na ordem parcial, é pelo menos N_v; e
  2. ela não vota em CS no estágio i.

É evidente por estas regras de "votação" que um processo pode determinar seus votos a partir das mensagens que ele recebe. Destes votos um processo pode decidir como e quando adicionar mensagens à ordem total. As regras de votação e os critérios de decisão juntos asseguram que todos os processos formam a mesma ordem total a partir da ordem parcial.

Os critérios de decisão

Os critérios de decisão para um processo P são:

No estágio i onde $i > 0$:

- P *decide por* CS se o número de mensagens na sua ordem parcial que votaram em CS no estágio i é pelo menos N_d, e para cada próprio subconjunto de CS, o processo decidiu contra este subconjunto.
- P *decide contra* CS se o número de mensagens em sua ordem parcial que votaram contra CS no estágio i é pelo menos N_d.

Uma vez que a decisão foi feita em favor de um conjunto candidato, as mensagens deste conjunto são incluídas na ordem total em uma seqüência arbitraria, mas determinística, e todo o procedimento é repetido com um novo conjunto de mensagens candidatas. Note que pela maneira como a ordem parcial é construída, um processo pode sempre determinar o voto de uma mensagem através da sua ordem parcial, já que todas as mensagens que precedem esta mensagem na ordem parcial deve ter sido recebida pelo processo. O protocolo Total assegura [PMMSA90] que: (1) se um processo decide por (contra) um conjunto candidato, então todos os processos decidem por (contra) este conjunto. (2) se um processo inclui um conjunto candidato como a sua j-ésima extensão à ordem total, então todos os processos incluem este conjunto nesta mesma ordem. Conseqüentemente, as ordens totais de todos os processos são a mesma. (3) a ordem total é consistente com a ordem parcial.

Os votos e decisões não precisam ser incluídos nas difusões; eles são deduzidos pelos processos através de seus reconhecimentos.

5.2.3 Desempenho

O desempenho do protocolo Trans/Total em termos de número de mensagens por cada difusão é dado pelo parâmetro N_d, isto é, número de votos para uma decisão. No geral, para um sistema resistente a k falhas, onde $k < n/3$ (n é o número de processos no grupo),

$$N_d = \frac{n + k + 1}{2}$$

5.2.4 Exemplo

Vamos ilustrar o protocolo Total através de exemplos. Primeiro, considere um sistema com seis processos, resistente a 1 falha. Neste sistema, quatro votos são necessários para uma decisão ($N_d = (6+2)/2 = 4$) de incluir um conjunto candidato na ordem total. Assuma que a transmissão é sem falha e todos os processos recebem todas as mensagens enviadas. Um exemplo da ordem parcial com cinco mensagens de tal sistema é uma cadeia linear, como visto na Figura 5.6.

O grafo com os reconhecimentos para todos os processo também será o mesmo, enquanto não há reconhecimentos negativos no sistema. No início, há apenas uma mensagem candidata,

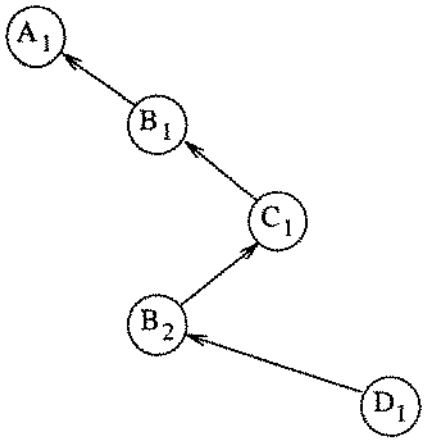


Figura 5.6: Exemplo de operação do Trans.

A_1, e assim apenas um conjunto candidato, $\{A_1\}$. As mensagens A_1, B_1, C_1 e D_1 votam para este conjunto candidato (B_2 não vota, já que B_1 o fez). Quatro votos são suficientes para uma decisão. Daí, no momento em que um processo recebe a mensagem D_1, ele pode decidir incluir A_1 na ordem total. Note que até D_1 ser recebida, um processo não pode decidir-se por A_1.

Agora, vamos considerar a ordem parcial vista na Figura 5.5. Assuma que há quatro processos no sistema. Daí, três votos ($N_d = (5+2)/2 = 3$) são necessários para uma decisão. As mensagens candidatas são B_1 e D_1 e os conjuntos candidatos são $\{B_1\}$, $\{D_1\}$, e $\{B_1, D_1\}$. Apenas as mensagens C_1 e B_1 votam em $\{B_1\}$, o que não é suficiente para uma decisão. Similarmente, A_1 e D_1 votam em $\{D_1\}$, o que também não é suficiente para uma decisão. Por esta razão, estes dois conjuntos candidatos são rejeitados. Para o conjunto candidato $\{B_1, D_1\}$, as mensagens C_1, A_1, D_2, e B_2 votam, o que é suficiente para uma decisão. Portanto, este conjunto candidato é escolhido para entrar na ordem total. Depois que estas mensagens são incluídas, as mensagens candidatas são A_1, C_1 e D_2. Pode-se observar que nenhum conjunto candidato destas mensagens conseguirá os três votos necessários. Assim, nenhuma inclusão na ordem total será feita até que novas mensagens sejam recebidas para alcançar o número de votos requeridos.

Um exemplo mais complexo é visto na Figura 5.7, para um sistema com seis processos. As mensagens geradas são: A_1, E_1, F_1, $D_1e_1f_1$, F_2e_1, C_1d_1, B_1a_1, E_2c_1, C_2b_1, $D_2c_2e_2$, A_2d_2 e B_2d_2. O sistema também é resistente a 1 falha: $N_d = 4$ e $N_v = 3$.

Os conjuntos candidatos $\{A_1\}$, $\{E_1\}$ e $\{F_1\}$ obtém muitos votos contra no estágio 0 e, assim, não são postas em ordem total. Já o conjunto $\{E_1, F_1\}$ obtém quatro votos a favor no estágio 0 das mensagens D_1, C_1, E_2 e F_2, o que é suficiente para uma decisão favorável. Mesmo se a mensagem F_2 for perdida, permanecerá três mensagens a favor no estágio 0, e quatro mensagens favoráveis no estágio 1 das mensagens E_2, D_2, A_2 e B_2. Novamente, isto seria suficiente para uma decisão favorável.

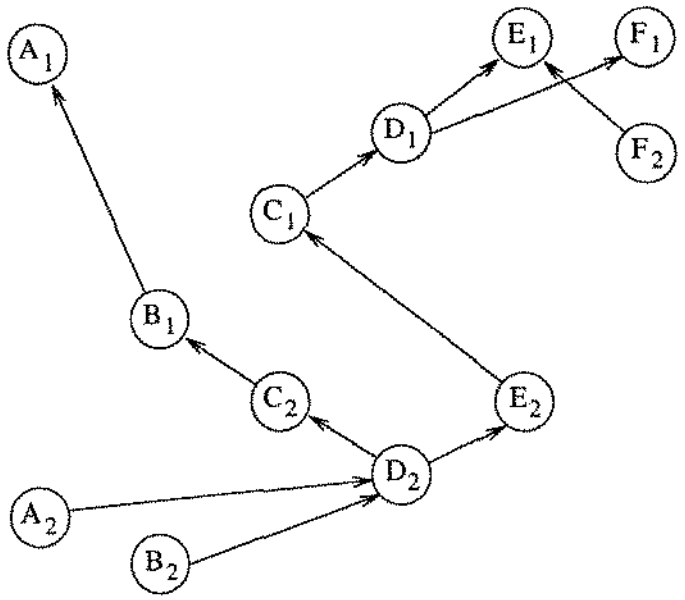


Figura 5.7: Um outro exemplo de operação do Trans.

5.3 Transis

O Sistema Transis [YAM92b, YAM92a, Kra93] implementa os protocolos chamados *Lansis* e *ToTo* para fornecer difusão confiável de mensagens. O protocolo Lansis é usado para entregar mensagens em ordem parcial e é derivado do protocolo Trans. A diferença principal entre o Lansis e o Trans está nos reconhecimentos. Um processo executando o protocolo Lansis espera para reconhecer mensagens até que elas possam ser entregues em ordem causal. A vantagem de esperar para reconhecer uma mensagem é que a ordem causal é diretamente definida pelos reconhecimentos.

Um processo executando o protocolo ToTo computa a ordem total das mensagens usando a ordem causal gerada pelo Lansis, com a troca adicional de mensagens para a votação da maioria na ordem das mensagens.

5.3.1 O Protocolo Lansis

Assim como o protocolo Trans [PMMSA90], o protocolo Lansis [YAM92b] explora a capacidade de *broadcast* de rede para disseminar mensagens de difusão através de uma simples transmissão. Assim, o protocolo Lansis também se baseia no princípio de que as mensagens serão ouvidas por todos os processos pertencentes ao grupo. O Lansis também usa um mecanismo de reconhecimento positivo e negativo estampado nas mensagens de difusão a fim de garantir a entrega de mensagens para todos do grupo.

Cada processo transmite mensagens de difusão com timestamps crescentes, servindo como identificadores destas mensagens. Por exemplo, o processo P_A envia $A_1, A_2, A_3, ...$ Um ACK (reconhecimento) consiste dos últimos timestamps das mensagens entregues por um processo,

e os ACK's são passados no cabeçalho das mensagens de difusão. As mensagens que seguem na ordem formam uma "cadeia" de ACK's, de forma que cada mensagem reconhece a anterior nesta cadeia, como na seqüência:

$$A_1, A_2, a_2B_1, B_2, B_3, b_3C_1, ...$$

Observe que, na notação do Lansis, o identificador de reconhecimento aparece antes da mensagem que o carrega, diferentemente do Trans. Mas, assim como no Trans, a detecção e a recuperação das mensagens perdidas é feita a partir da análise da cadeia de reconhecimento das mensagens já recebidas. Um processo pode, então, emitir um *ACK-negativo* de uma mensagem perdida, pedindo sua retransmissão. As mensagens entregues são guardadas em um mecanismo de armazenamento estável, de forma que os pedidos de retransmissão podem ser atendidos por qualquer dos processos participantes do grupo.

No Lansis, uma mensagem nova contém ACK's para todas as mensagens (não-reconhecidas) causalmente entregáveis. Esta é a principal diferença entre o Trans e o Lansis, onde ACKs no Lansis reconhecem a entregabilidade das mensagens ao invés das suas recepções. Assim, eles refletem a relação causa e efeito nos processos *diretamente*. No Trans, por outro lado, a ordem parcial não corresponde à ordem dos eventos de um processo e é obtida aplicando-se o predicado OPD nos reconhecimentos. Além disto, os critérios de entrega no Lansis é significantemente simplificado por esta modificação.

Assim como introduzido no Psync, a ordem parcial pode ser representada através de um grafo direcionado acíclico: os nós são as mensagens e as arestas conectam duas mensagens que estão diretamente dependentes na ordem causal. O Grafo Direcionado Acíclico Causal, ou simplesmente Grafo Causal (**GC**), contém todas as mensagens enviadas no sistema. Os processos vêem o mesmo grafo, apesar de que em determinados instantes os grafos de cada processo podem estar diferentes.

Implementação dos Serviços

Os serviços de difusão são fornecidos pela entrega de mensagens que pertencem ao Grafo Causal. Estes serviços diferem pelos critérios que determinam quando entregar as mensagens pertencentes ao Grafo Causal. Estes critérios operam com a estrutura do Grafo e não envolvem considerações externas como tempo, atrasos, etc.. Os serviços de entrega são os seguintes:

1. Básico: Entrega imediata.
2. Causal: Quando todas as predecessoras no Grafo já foram entregues.
3. Concordado: Tratado pelo protocolo ToTo, que entrega as mensagens em ordem total.
4. Seguro: Quando os caminhos da mensagem para as folhas do Grafo contém uma mensagem de cada processo.

Observe que o serviço de entrega seguro representa a mesma abordagem do protocolo Psync para a determinação e entrega de mensagens estáveis – uma mensagem m está estável (ou, neste protocolo, está *segura*) quando todos os processos difundem alguma outra mensagem que reconhece direta ou indiretamente m.

5.3.2 O Protocolo ToTo

O protocolo ToTo [Kra93] implementa o serviço de difusão concordada. As entradas para o protocolo ToTo são eventos de dois tipos:

1. Mensagens: uma cadeia de mensagens causalmente ordenadas. Toda mensagem é unicamente identificada por $< remetente, contador >$. Denotaremos como $m_{p,i}$ a i-ésima mensagem enviada pelo processo p. A ordem causal é representada por um Grafo Causal dinâmico, envolvendo todas as mensagens que são trocadas no grupo. Denotaremos como GC_p o Grafo Causal do processo p.

2. $Falha(q)$: evento gerado pelo protocolo de Gerenciamento do Grupo, que indica que o processo q falhou e nenhuma mensagem de q será recebida até que q retorne ao grupo (usando o protocolo de Gerenciamento do Grupo). Todos os processos recebem o mesmo conjunto de mensagens do processo q antes de receber $Falha(q)$. Note que $Falha(q)$ é um evento interno, e não está associado com uma mensagem e nem é inserido no GC.

O protocolo ToTo recebe do protocolo Lansis uma cadeia de mensagens causalmente ordenadas através do Grafo Causal, que é incrementalmente revelado para os processos, sempre contendo todas as mensagens trocadas no grupo, ou seja, não contém "buracos" de mensagens perdidas. Mais formalmente, se $m \in GC_p$ então para toda mensagem m', tal que m segue m' na ordem causal, então $m' \in GC_p$.

Assim como no desenvolvimento do Total, o principal objetivo de projeto do ToTo foi desenvolver um protocolo que permitisse alcançar consistência global no grupo de comunicação, baseado apenas nas informações disponíveis localmente (representadas pelo GC_p). A principal dificuldade nisto é que, devido a perda de mensagens, todo processo tem apenas uma visão subjetiva do estado do GC global. O protocolo deve evitar uma decisão até que o estado local de GC_p assegure consistência global. Uma maneira de assegurar consistência global é atrasar a entrega de mensagens de difusão concordadas até que todos os processos do grupo tenham reconhecido as suas recepções. Infelizmente, esta solução simples pode trazer um considerável custo na latência das mensagens. O protocolo ToTo foi projetado para reduzir esta latência conseguindo um limite de tempo para o pior caso. A idéia principal é usar a ordem causal para acelerar as decisões.

Notação e Definições

Uma mensagem é dita *pendente* se ela já foi recebida mas não foi inserida na ordem total, e assim, não foi entregue. Assim como no Total, uma mensagem pendente, que segue na ordem causal

apenas mensagens já entregues, é chamada de *mensagem candidata*. O conjunto de mensagens candidatas é chamado de *conjunto candidato*. Este conjunto é equivalente ao do protocolo Total e ao conceito de "ondas" do Psync. Um exemplo deste conjunto é visto na Figura 5.8.

Seja $Defunt_p$ o conjunto de todos os processos, q, para os quais p recebeu um evento $Falha(q)$ e q ainda está no grupo. Seja $M_p = \{m_1, ..., m_k\}$ o conjunto de mensagens candidatas no processo p. Associaremos com cada mensagem m_i as funções $Tail_p$, $NTail_p$, VT_p e NVT_p:

1. $Tail_p(m_i) = \{q \mid \exists m_{q,j}$ tal que $m_{q,j} \in GC_p$, $m_{q,j}$ segue $m_i\}$. $Tail_p(m_i)$ é o conjunto de todos os processos que enviaram uma mensagem causalmente seguinte a m_i no GC_p corrente.

2. $NTail_p(m_i) = \mid Tail_p(m_i) \mid$ e $NTail_p = \mid Tail_p \mid$.

3. Toda mensagem candidata m_i é associada com um *Vetor de Timestamp Estendido* (VTE), $VT_p(m_i)$. O VTE tem uma entrada para cada processo do grupo. Seja q um processo do grupo e $m_{q,k}$ a primeira mensagem de q (se existe) que segue causalmente qualquer mensagem candidata m_i:

$$VT_p(m_i)[q] = \begin{cases} * & \text{se } q \notin Tail_p \\ k & \text{se } m_{q,k} \text{ segue } m_i \\ \infty & \text{caso contrário} \end{cases}$$

4. $NVT_p(m_i) = \mid \{q \mid VT_p(m_i)[q] \neq *, VT_p(m_i)[q] \neq \infty\} \mid$.

Em uma notação compacta, usaremos $Tail_{pi} \equiv Tail_p(m_i)$, $NTail_{pi} \equiv NTail_p(m_i)$, $VT_{pi} \equiv VT_p(m_i)$, e $NVT_{pi} \equiv NVT_p(m_i)$.

As funções acima são dinamicamente computadas em cada processo toda vez que uma mensagem chega e é inserida localmente no GC. Note que o GC local é um parâmetro para todas estas funções. Na seção 5.3.4 são apresentados os valores destas funções para o exemplo de GC visto na Figura 5.8.

Seja $\Phi \geq \frac{n}{2}$ um parâmetro limiar. Usando VT_p, $Tail_p$ e $NTail_p$, definiremos duas funções de comparação, $Vence_p$ e $Futuro_p$:

$$Vence_p(VT_{p1}, VT_{p2}) = \begin{cases} 1 & \text{se } \mid \{j \mid VT_{p1}[j] < VT_{p2}[j]\} \mid > \Phi \\ 0 & \text{se } \mid \{j \mid VT_{p2}[j] < VT_{p1}[j]\} \mid > \Phi \\ \chi & \text{caso contrário} \end{cases}$$

É fácil verificar que para $\Phi \geq \frac{n}{2}$:

$$Vence_p(VT_{p1}, VT_{p2}) = 1 \Rightarrow \forall i Vence_p(VT_{p2}, VT_{pi}) \neq 1$$

Quando as mensagens chegam e são inseridas em GC_p, $Vence_p$ muda. $Vence_p$ pode mudar seu valor de χ para 1 ou 0. Em determinado estágio, $Vence_p$ fixa seu valor permanentemente. A

última mudança pode ser quando $NTail_p = n$. Definiremos $Futuro_p(VT_{p1}, VT_{p2})$ como sendo o conjunto de todos os valores possíveis permanentes de $Vence_p(VT_{p1}, VT_{p2})$.

O valor de $Futuro_p$ pode intuitivamente ser pensado como o conjunto de todos os futuros valores de comparação possíveis. Dado VT_{p1} e VT_{p2}, $Futuro_p$ é determinado como:

$$\begin{aligned} 1 \in Futuro_p(VT_{p1}, VT_{p2}) &\Leftrightarrow |\{j \mid VT_{p2}[j] \leq VT_{p1}[j]\}| < n - \Phi \\ \chi \in Futuro_p(VT_{p1}, VT_{p2}) &\Leftrightarrow Vence_p(VT_{p1}, VT_{p2}) = \chi \\ 0 \in Futuro_p(VT_{p1}, VT_{p2}) &\Leftrightarrow 1 \in Futuro_p(VT_{p2}, VT_{p1}) \end{aligned}$$

O Algoritmo

O fluxo do protocolo ToTo pode ser visto como uma série de *ativações*. Em toda ativação do protocolo um conjunto de mensagens candidatas são consideradas para entrega em ordem total. Uma ativação está completa quando um subconjunto de mensagens candidatas é entregue. Assim, os números das ativações servem como ordem total de entrega das mensagens. O protocolo ToTo usa a estrutura do GC_p para definir uma função de quantificação no conjunto de mensagens candidatas. Esta função, $VT_p(m)$ associa cada mensagem candidata com um *vetor de timestamp*. O vetor de timestamp tem uma entrada para cada processo do grupo. Usando VT_p, define-se $Vence_p(VT_p(m), VT_p(m'))$, que pode ter três valores: 1 (vence), 0 (perde) e χ (empate). $Vence_p$ define a relação corrente entre cada par de mensagens candidatas. Usando esta relação, define-se o conjunto "mínimo" atual de mensagens candidatas, chamado de mensagens *fontes*. Uma mensagem fonte é uma mensagem candidata que é "igual" ou "menor" que todas as outras mensagens candidatas de acordo com a relação definida por $Vence_p$. A entrega de um conjunto de mensagens fontes é retardada até que o protocolo alcance um *estado estável*. Informalmente, um estado é estável sempre que os *critérios de entrega*, descritos a seguir, são encontradas.

Uma mensagem candidata m é uma mensagem *fonte* se, para qualquer outra mensagem m', $1 \notin Futuro_p(VT_p(m'), VT_p(m))$. Seja S_p o conjunto de mensagens fontes em p. Os critérios para entrega das mensagens são os seguintes, com $\frac{n}{2} \leq \Phi < n$:

(DEL) Entregue S_p quando:

1. Estabilidade interna: o conjunto de mensagens fontes não pode mais aumentar nesta ativação com qualquer outra mensagem candidata. Formalmente,
   $\forall m \in M_p - S_p$ e $\exists m' \in M_p$ tal que $Futuro(VT_p(m'), VT_p(m)) = \{1\}$. E,
2. Estabilidade externa: o conjunto de mensagens fontes não pode mais aumentar nesta ativação com qualquer outra mensagem "escondida" (não vista). Formalmente:
   (a) $\exists s \in S_p$ t.q. $NVT_p(s) > \Phi$ e $\forall s \in S_p$ t.q. $NTail_p(s) \geq n - \Phi$
   Ou,
   (b) $NTail_p = n$. Ou seja, todos os processos enviaram alguma mensagem que segue alguma mensagem candidata.

Este procedimento é conhecido como protocolo $ToTo_\Phi$ e é descrito no Algoritmo 5.3. Ele especifica como tratar as mensagens que chegam, seus critérios de entrega e o efeitos tomados quando estas mensagens são entregues. O contador $Nact$ é usado para indexar ativações sucessivas. Ele é incrementado em cada entrega e é reiniciado toda vez que o grupo é reconfigurado.

Usando o protocolo ToTo a latência na distribuição das mensagens depende diretamente do parâmetro Φ. Aumentar Φ pode reduzir a latência das mensagens somente quando o tempo de espera para o critério (DEL) 2(a) é dominado pela condição:

$$\forall s \in S_p \mid NTail_p(s) \geq n - \Phi$$

Intuitivamente, isto acontecerá quando $k = \mid S_p \mid$ é grande. Para um k grande, para todas as mensagens candidatas o tempo de espera para coletar $n-\Phi$ reconhecimentos ($NTail_p(s) \geq n-\Phi$ para toda mensagem fonte s) dominará o tempo de espera para qualquer mensagem candidata coletar os Φ primeiros reconhecimentos ($NVT_p(s) > \Phi$ para alguma mensagem candidata s). Para sistemas de comunicação em grupo com poucos computadores conectados em uma rede local simples, o Φ ótimo é $\frac{n}{2}$.

5.3.3 Desempenho

O custo em latência na entrega das mensagens de difusão dos vários serviços do Transis, é apresentado na Tabela 5.2.

O custo é dado pelo número de mensagens necessárias para a entrega de uma mensagem de difusão, incluindo a própria mensagem. n é o número de processos no grupo. Na entrega causal, dada uma mensagem m, se todas as suas predecessoras já foram recebidas e entregues, a sua entrega será imediata, assim como mensagens do tipo de serviço básico, que não dependem de qualquer outra mensagem.

Na entrega concordada, o custo é dado pelo parâmetro Φ informado ao protocolo ToTo. Como discutido anteriormente, o valor ótimo de Φ é $n/2$. Assim, com a própria mensagem, o custo de uma difusão é $n/2 + 1$.

O tipo de serviço seguro somente entrega uma mensagem de difusão quando esta já foi recebida por todos os processos e estes já enviaram mensagens que serviram como reconhecimentos destas recepções. Portanto, o custo é n.

5.3.4 Exemplo

A Figura 5.8 apresenta uma representação de um possível estado de GC_p. As mensagens de difusão concordadas (em ordem total) são denotadas por círculos hachurados, enquanto que mensagens em ordem causal são denotadas por círculos brancos. As mensagens $m_{e,1}$, $m_{a,1}$ e $m_{f,1}$ estão entregues. O conjunto candidato é $\{m_{b,1}, m_{c,1}, m_{f,2}\}$, enquanto $m_{c,2}$ é uma mensagem pendente que, como ela segue $m_{c,1}$, não é uma mensagem candidata.

A Tabela 5.3 apresenta os valores das funções para o exemplo de GC_p visto na Figura 5.8.

Ativação #Nact no processo p:

Ao receber uma mensagem $m_{q,k}$:

se $m_{q,k}$ é uma nova mensagem candidata então 1
  para todo processo $r \in Tail_p$ faça 2
    $VT_p(m_{q,k})[r] := \infty$; 3
  fim_para
fim_se
insira $m_{q,k}$ em M_p; 4
se $q \notin Tail_p$ então 5
  para toda $m_i \in M_p$ tal que $m_{q,k}$ segue m_i faça 6
    $VT_{pi}[q] := k$; 7
  fim_para
fim_se
para toda $m_i \in M_p$ não seguida por $m_{q,k}$ faça 8
  $VT_{pi}[q] := \infty$; 9
fim_para

Ao receber um evento $Falha(q)$:

se $q \notin Tail_p$ então 10
  para toda $m_i \in M_p$ faça 11
    $VT_{pi}[q] := \infty$; 12
  fim_para
fim_se
insira q em $Defunt_p$; 13

Quando os critérios (DEL) são satisfeitos:

entregue todas as mensagens em S_p na ordem lexicográfica; 14
atualize M_p; 15
para toda m em M_p faça 16
  recalcule $VT_p(m)$; 17
fim_para
$Nact := Nact + 1$; 18

Na reconfiguração do grupo:

$Nact := 1$; 19

Algoritmo 5.3: Protocolo $ToTo_\phi$.

Tipo de serviço	*Índice de sincronismo*
Seguro	n
Concordado	$n/2+1$
Causal	1
Básico	1

Tabela 5.2: Hierarquia de Serviços do Transis.

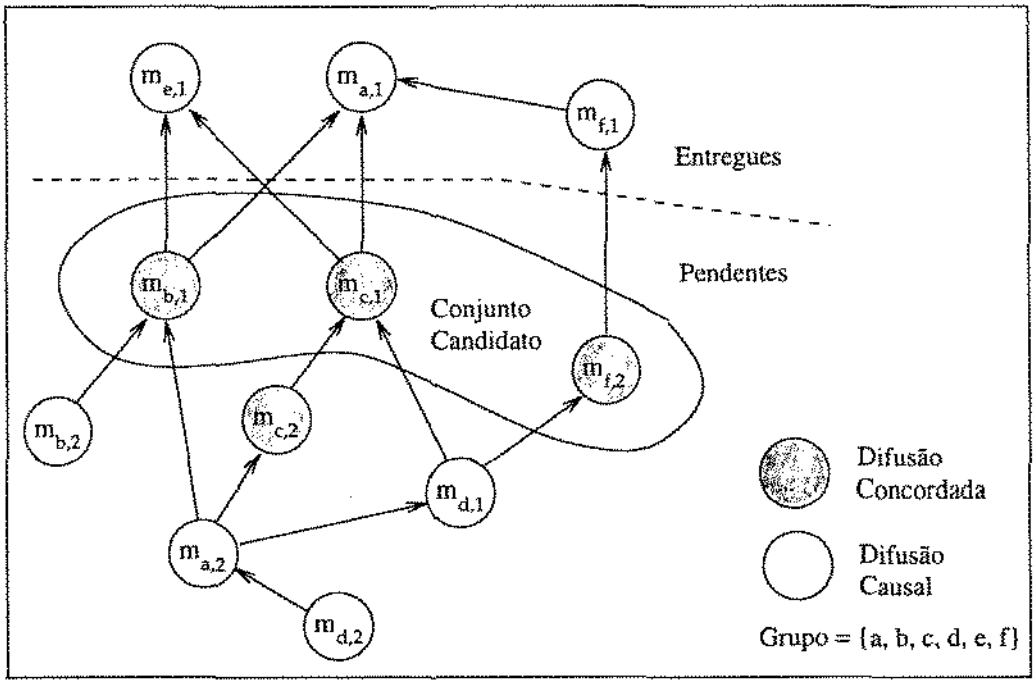


Figura 5.8: Uma representação do GC_p.

$Tail_p(m_{b,1}) = \{a, b, d\}$	$NTail_p(m_{b,1}) = 3$
$Tail_p(m_{c,1}) = \{a, c, d\}$	$NTail_p(m_{c,1}) = 3$
$Tail_p(m_{f,2}) = \{a, d, f\}$	$NTail_p(m_{f,2}) = 3$
$Tail_p = \{a, b, c, d, f\}$	$NTail_p = 5$
$VT_p(m_{b,1}) = (2, 1, \infty, \infty, *, \infty)$	$NVT_p(m_{b,1}) = 2$
$VT_p(m_{c,1}) = (2, \infty, 1, 1, *, \infty)$	$NVT_p(m_{c,1}) = 3$
$VT_p(m_{f,2}) = (2, \infty, \infty, 1, *, 2)$	$NVT_p(m_{f,2}) = 3$

Tabela 5.3: Valores das funções para GC_p.

5.4 Comentários

O protocolo Psync tem sido implementado como parte do sistema operacional x-kernel. A responsabilidade de fornecer ordenação total e consistência entre grupos de comunicação é deixada para o programa da aplicação. Vários serviços foram adicionados ao protocolo Psync, tais como o algoritmo de gerenciamento do grupo [SMS91]. Usando este algoritmo, os processos conseguem eventual concordância nas mudanças da configuração do sistema. O algoritmo trata de falhas de processos e permite que um processo se una a um grupo já existente. Partições e re-junções de rede não são suportadas.

Os protocolos Trans e Total mantém causalidade e asseguram que processos ativos (operacionais) continuam a ordenar mensagens mesmo quando outros processos falharam, contanto que um requisito de resiliência seja satisfeito. Um protocolo de gerenciamento do grupo [LEMA94a] é implementado no topo do protocolo Total. Se um processo "suspeita" de um outro processo, ele envia uma mensagem de falha para o processo suspeito. Quando esta mensagem é ordenada, o grupo é alterado para excluir este processo. A limitação desta arquitetura é que se o Total não pode ordenar as mensagens de mudanças na configuração do grupo (quando, por exemplo, as restrições de resiliência não são satisfeitas), o sistema fica bloqueado.

No protocolo Transis, dois algoritmos para ordenação total estendem a ordem causal para uma ordem concordada total. O primeiro é o algoritmo *todos-reconhecem*, que é similar ao algoritmo usado no Psync, onde uma mensagem só se torna estável quando todos os processos enviam alguma informação garantindo que não há nenhuma outra mensagem dependente causalmente desta primeira. O segundo protocolo é o ToTo, descrito neste trabalho. Ambos computam a ordem total baseados no grafo de contexto sem precisar da troca de mensagens adicionais. Enquanto o ToTo é mais eficiente que o protocolo *todos-reconhecem*, ele não pode manter sincronismo virtual estendido.

O algoritmo de gerenciamento do grupo do Transis [YAM92a] é um protocolo simétrico que foi o primeiro a tratar de partição e re-junção de redes. Embora operacional em ambientes assíncronos, o algoritmo assegura término em um limite de tempo. A idéia básica deste algoritmo de gerenciamento foi adotada pelos sistemas Totem e Horus. Uma referência mais completa sobre o Transis e seu algoritmo de gerenciamento do grupo pode ser encontrada em [Mal94].

Capítulo 6

Outros Protocolos

Este Capítulo descreve alguns protocolos que não se enquadram na classificação que definimos para diferenciar as estratégias utilizadas nas difusões. O primeiro protocolo apresentado, o de Garcia-Molina e Spauster, também utiliza uma estrutura de grafo mas, diferentemente dos outros modelos, o grafo determina o fluxo de mensagens entre os vários processos de diferentes grupos a fim de determinar a ordem das mensagens entre os vários grupos.

O segundo protocolo descrito neste Capítulo é o de Drummond e Babaöglu, que utiliza redundância nos canais de comunicação da rede a fim de suportar um certo número de falhas. O algoritmo desenvolvido é síncrono.

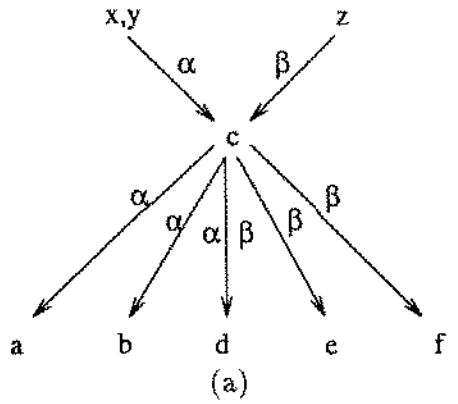
6.1 Protocolo de Garcia-Molina e Spauster

Garcia-Molina e Spauster [GMS91] propõem uma nova solução para garantir ordenação de mensagens entre múltiplos grupos, em um ambiente de difusão, chamada de *algoritmo do Grafo de Propagação*. Este algoritmo é baseado no de Chang e Maxemchuk, no que diz respeito à centralização no controle das ordenações, tentando, desta forma, reduzir a sobrecarga dos algoritmos que usam soluções totalmente distribuídas. Entretanto, ao invés de ordenar todas as mensagens em um processo centralizador, elas são ordenadas por uma coleção de processos estruturados em um *grafo de propagação de mensagens*.

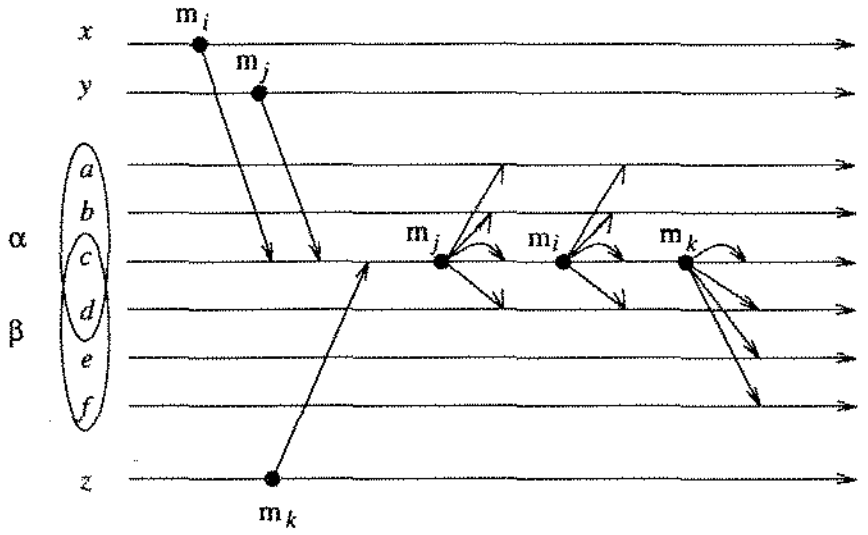
Cada vértice no grafo representa um processo. O grafo indica o caminho que a mensagem deve seguir para alcançar todos os membros do grupo. Ao invés de enviar diretamente as mensagens para os seus destinos e depois ordená-las, as mensagens são propagadas via uma série de processos que as ordenam ao longo do caminho, fazendo a união das mensagens endereçadas para diferentes grupos. Quando um destes processos recebe uma mensagem, ele sabe imediatamente em que ordem a mensagem deverá estar. A idéia básica é usar processos que são interseções de grupos de difusão como vértices intermediários que realizam a ordenação das mensagens relativa aos grupos.

Para ilustrar, considere o seguinte exemplo. Sejam x, y e z processos não pertencendo a qualquer grupo de comunicação. Os processos x e y estão enviando mensagens para o grupo

$\alpha = a, b, c, d$, enquanto que o processo z está enviando para o grupo $\beta = c, d, e, f$. Os processos pertencentes a estes dois grupos são denotados por a, b, c, d, e e f. Suponha que x envia para o grupo α a mensagem m_i com o timestamp T_1. Quando o processo c recebe m_i ele não pode imediatamente entregá-la para os processos destinos. Ele primeiro deve descobrir de todas as fontes potenciais se há alguma outra mensagem com um timestamp menor.



(a)



(b)

Figura 6.1: (a) Grafo de propagação dos grupos α e β. (b) O processo c une e ordena as mensagens de α e β.

Usando a situação descrita acima, um simples grafo de propagação para este problema é indicado na Figura 6.1(a). A fim de garantir que todas as mensagens são entregues na mesma ordem relativa, x, y e z enviam suas mensagens apenas para c, que "une", compara e ordena estas mensagens. Como visto na Figura 6.1(b), o processo c repassa as mensagens do grupo α vindas de x e y para a e b, e as mensagens do grupo β vindas de z para e e f, e une e ordena as mensagens de α e β para ele mesmo e para d. Assim, todos os processos entregam suas mensagens na ordem definida por c.

Este algoritmo tem dois componentes: o gerador do grafo de propagação (PG) e o protocolo de passagem de mensagens (MP). O gerador PG constrói o grafo de propagação para um determinado conjunto de grupos de difusão. O grafo gerado é então passado para os processos para que estes usem o protocolo MP para enviar, receber, propagar e repassar as mensagens. Os processos fontes enviam suas mensagens de difusão para um processo do grupo, chamado de *destino primário*, que é o processo que aparece em vários grupos e realiza a ordenação total das mensagens.

O Gerador PG

Este algoritmo deve garantir as seguintes propriedades:

1. Se as mensagens m_1 e m_2 são enviadas para dois processos, então elas são entregues na mesma ordem relativa, caso haja interseções entre os grupos de comunicação e elas vieram de diferentes transmissores e são endereçadas para diferentes receptores.

2. Se x está no grupo α, então x recebe todas as mensagens destinadas a α.

Para satisfazer estes requisitos, é necessário que o grafo de propagação tenha as seguintes propriedades:

- (PG1) Para cada grupo α há um único destino primário p; e
- (PG2) Para cada processo $x \in \alpha$, há um único caminho de p para x.

Há também duas propriedades adicionais que o grafo pode exibir e que o gerador PG tenta fornecer:

- (PG3) O destino primário do grupo α é um membro de α; e
- (PG4) Seja p o destino primário de α e x um outro processo em α. Então, os nós no caminho de p para x são todos membros de α.

Quando existe um nó a no caminho de p até x e a não é membro de α, então a é chamado de *nó extra*. O grupo que contém a raiz é chamado de *grupo raiz* e os processos do grupo raiz são chamados de *interseções*.

Ao iniciar, o gerador PG seleciona o processo que aparece o maior número de vezes entre os grupos e faz dele a raiz. Para determinar os filhos da raiz, o procedimento *new_subtree* é chamado passando a raiz como parâmetro. Este procedimento particiona os grupos que não contém a raiz de forma nenhum grupo intercepta um outro em uma outra partição, ou seja, não tenha processos em comum. A fim de alcançar a propriedade PG4, o gerador apenas considera aquelas partições que contenham uma *interseção*. A partir disto, uma das interseções é escolhida como o filho da raiz, usando a mesma heurística usada para escolher a raiz: escolha o processo que aparece na maioria dos grupos. Finalmente, podem haver processos que são interseções, mas que não ocorrem em qualquer partição. Estes processos tornam-se, então, filhos da raiz.

Para gerar o próximo nível da árvore, uma chamada recursiva a *new_subtree* é feita para cada filho, passando-o como parâmetro. O ciclo continua até que não haja mais processos restantes.

O Protocolo MP

O destino primário para cada grupo de difusão é o membro do grupo mais próximo da raiz. Um processo que recebe uma mensagem propaga-a para as subárvores abaixo que contém os membros do grupo destino da mensagem.

O protocolo MP requer que cada processo mantenha um número de seqüência para cada processo ao qual ele envia uma mensagem, como determinado pelo grafo de propagação. Isto garante que um receptor possa ordenar corretamente as mensagens de um transmissor no caso destas chegarem fora de ordem. Isto permite também a detecção de mensagens perdidas. Reconhecimentos não são requeridos no protocolo MP; mensagens nulas e *timeouts* são usados para a detecção de falhas.

Em cada processo duas filas são mantidas: uma fila para as mensagens destinadas para o processo local e uma fila de espera para as mensagens que chegam fora de seqüência. Quando um processo recebe uma mensagem, ele verifica o número desta com o número de seqüência que ele esperava receber daquele processo. Se estes números não são iguais, a mensagem é posta na fila até que uma mensagem mais antiga seja recebida. Caso contrário, o processo repassa a mensagem para os seus descendentes na árvore que pertencem ao seu grupo usando o número de seqüência apropriado para cada filho da subárvore.

O Algoritmo

Por razões de simplicidade na explicação do algoritmo, consideraremos que todos os processos na árvore do grafo de propagação são membros do mesmo grupo D, assim todas as mensagens "entram" na árvore pela raiz. O algoritmo tolera falhas de parada dos processos e perda ou recebimento fora de ordem de mensagens. Assume-se o seguinte:

A1 Há um mecanismo de armazenamento estável.

A2 A árvore lógica inicial é conhecida por todos os processos; em particular, todos os processos sabem quem são seus pais e quem são seus filhos nesta árvore.

A3 Uma mensagem m é designada por $[s, D, n]$, onde s é a identificação do processo origem, D é o grupo destino e n é um número de seqüência. Este número de seqüência é atribuído pelo processo origem.

Todos os processos na árvore, exceto a raiz, mantém uma variável, L_p, para assegurar que mensagens de seu pai não estão faltando. L_p contém o timestamp da última mensagem em ordem recebida do processo pai na árvore. Cada processo pai, incluindo a raiz, mantém uma variável L_c para estampar as mensagens enviadas para seus filhos. Ainda, cada processo da árvore mantém um vetor local $filhos(D)$, que informa quais processos no grupo D são filhos deste processo. A raiz mantém uma variável L_s^i para os timestamps das mensagens de cada processo fonte i (atribuído de acordo com [A3]). Em todos os processos, as mensagens são mantidas em um LOG para propósitos de recuperação. O LOG deve estar em um armazenamento estável. Uma

fila, Q, para mensagens não entregues também é usada. Os processos fontes (origem) devem manter um número de seqüência L_r, para estampar as mensagens para a raiz. L_r é iniciada com 0. Os processos fontes ainda devem armazenar as mensagens que eles enviam em um LOG também estável. Cada fonte mantém a variável $raiz(D)$, que informa a identificação da raiz na árvore para o grupo D.

O pseudocódigo para o protocolo é visto no Algoritmo 6.1. O algoritmo funciona como segue. Quando um processo fonte, s, deseja difundir uma mensagem $m = [s, D, n]$ para o grupo D, ele primeiro incrementa sua variável local L_r, armazena a mensagem no LOG e a envia para o destino primário de D, neste caso a raiz. Ao receber a mensagem, o destino primário, denotado por r, verifica n para ver se esta é a próxima mensagem que ele espera receber desta fonte (n é comparado com L_s^s). Se é, então r incrementa L_s^s, incrementa L_c, armazena a mensagem, entrega-a localmente e a envia para cada filho apropriado, com L_c adicionado à mensagem. Ele então verifica Q para ver se há mensagens pendentes que podem ser entregues. Para cada mensagem que pode ser entregue, ele processa como descrito acima, incrementando primeiro L_s^s.

Se a mensagem recebida de s não é a próxima mensagem esperada de s, então há duas possibilidades. Ou a mensagem é mais antiga que L_s^s, e neste caso a mensagem é uma duplicata que é descartada; ou a mensagem é posta em Q e r requisita as mensagens faltantes com os timestamps entre L_s^s e n.

Quando um membro não-raiz do grupo, q, recebe uma mensagem $([s, D, n], x)$ do seu pai, ele procede quase da mesma forma que o destino primário. A principal diferença é que q verifica x, o número adicionado pelo pai, com sua variável local L_p para ver se esta é a próxima mensagem que ele espera receber de seu pai. Quando q propaga a mensagem para seus filhos, ele substitui x pelo seu valor local de L_c.

6.1.1 Corretude

Para mostrar que a árvore gerada pelo protocolo MP garante ordenação de múltiplos grupos, devemos provar duas propriedades: (1) todos os processos entrega as mensagens recebidas na mesma ordem; (2) todos os processos recebem as mensagens destinadas ao grupo.

Para mostrar a propriedade 1, considere dois processos a e b, que recebem as mensagens m_1 e m_2. Digamos que a entrega estas na ordem m_1m_2 e b entrega-as na ordem m_2m_1. Se m_1 e m_2 são mensagens para o mesmo grupo de difusão α, então inicialmente elas são ordenadas pelo processo destino primário de α. É fácil ver que o esquema de timestamps usado no protocolo MP garante que esta ordem é mantida como m_1m_2 e que é propagada para todos os processos.

Suponha, então, que m_1 é destinada para os grupos α_1 e m_2 é destinada para α_2. Sejam $dp(\alpha_1)$ e $dp(\alpha_2)$ os destinos primários destes dois grupos, respectivamente. Se $dp(\alpha_1)$ e $dp(\alpha_2)$ são o mesmo processo, então a situação é a mesma como se m_1 e m_2 fossem destinadas para o mesmo grupo α_1. Digamos então que $dp(\alpha_1)$ e $dp(\alpha_1)$ são dois processos diferentes. Certamente, $dp(\alpha_1)$ e $dp(\alpha_2)$ são pais de a e b na árvore de propagação. Pela propriedade da árvore, sabemos que há apenas um caminho da raiz para qualquer nó. Assim, ou $dp(\alpha_1)$ é "pai" de $dp(\alpha_2)$ ou vice-versa. Digamos que $dp(\alpha_1)$ é pai de $dp(\alpha_2)$. Então, em $dp(\alpha_2)$, m_1 e m_2 são ordenadas e

Difusão:

A fonte s ao difundir uma mensagem $([s, D, n])$:

$n := L_r$; $L_r := L_r + 1$; LOG$([s, D, n])$; 1
envie $[s, D, n]$ para $raiz(D)$; 2

Recebimento e Entrega:

A raiz, ao receber $[s, D, n]$:

se $n = L_s^s + 1$ então 3
$L_s^s := L_s^s + 1$; $L_c := L_c + 1$; LOG$([s, D, n])$; 4
entregue a mensagem $[s, D, n]$; 5
para cada $g \in filhos(D)$ faça 6
envie $([s, D, n], L_c)$ para g; 7
enquanto há uma mensagem $[s, D, L_s^s + 1]$ em Q faça 8
remova a mensagem $[s, D, L_s^s + 1]$ de Q; 9
$L_s^s := L_s^s + 1$; $L_c := L_c + 1$; LOG$([s, D, L_s^s], L_c)$; 10
entregue a mensagem $[s, D, L_s^s]$; 11
para cada $g \in filhos(D)$ faça 12
envie $([s, D, L_s^s])$ para g; 13
fim_enquanto
senão se $n > L_s^s + 1$ então 14
insira $[s, D, n]$ em Q; 15
para toda mensagem $[s, D, x]$ tal que $[s, D, x] \notin Q$ e $L_s^s < x < n$ faça 16
requisite retransmissão de $[s, D, x]$ a s; 17
senão ignore a mensagem; 18

O membro não-raiz a ao receber $([s, D, n], p)$ de b, pai de a:

se $p = L_p + 1$ então 19
$L_p := L_p + 1$; $L_c := L_c + 1$; LOG$([s, D, n], L_c)$; 20
entregue a mensagem $([s, D, n], p)$; 21
para cada $g \in filhos(D)$ faça 22
envie $([s, D, n], L_c)$ para g; 23
enquanto há uma mensagem $([s, D, x], L_p + 1)$ em Q faça 24
remova a mensagem $([s, D, x], L_p)$ de Q; 25
$L_p := L_p + 1$; $L_c := L_c + 1$; LOG$([s, D, x], L_c)$; 26
entregue a mensagem $([s, D, x])$; 27
para cada $g \in filhos(D)$ faça 28
envie $([s, D, x], L_c)$ para g; 29
fim_enquanto
senão se $p > L_p + 1$ então 30
insira $([s, D, n], p)$ em Q; 31
fim_se

Algoritmo 6.1: Protocolo MP de Garcia-Molina e Spauster.

propagadas para a e para b. Já que facilmente podemos observar que esta ordem é preservada pelo protocolo MP, a e b não podem entregar estas mensagens em uma ordem inconsistente.

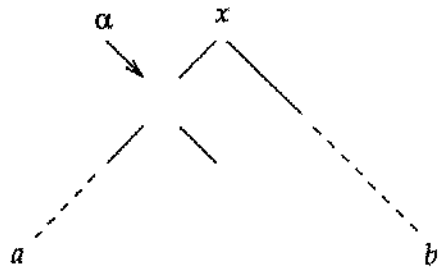


Figura 6.2: Inconsistência na árvore de propagação.

Não é difícil, também, ver que o algoritmo garante que todos os processos recebem as mensagens endereçadas a eles. Digamos que algum processo não recebe alguma mensagem destinada a ele. Esta situação deve parecer como visto na Figura 6.2, onde a e b são supostos a receberem mensagens do grupo α, mas b não está em um caminho para as mensagens de α. A Figura 6.2 indica que o gerador PG põe os nós a e b em diferentes subárvores de x, embora eles sejam do mesmo grupo α. Isto é uma impossibilidade visto que a rotina de particionamento *new_subtree* põe todos os processos do mesmo grupo em uma mesma subárvore.

6.1.2 Desempenho

Para avaliar o desempenho do protocolo, observou-se a utilização de mensagens *broadcast* e de mensagens ponto-a-ponto para o algoritmo de propagação. A Tabela 6.1 indica o desempenho do protocolo para a medida N, número de mensagens requeridas para realizar uma difusão sobre a propriedade de ordenação entre múltiplos grupos.

	ponto-a-ponto	*broadcast*
N	n + ϵ	d + 1

Tabela 6.1: Desempenho do protocolo de propagação de Molina e Spauster

Claramente, o desempenho do algoritmo de propagação depende dos valores de ϵ e d. Onde ϵ é o número esperado de *nós extras*, e d é o número de níveis (profundidade) da árvore. Através de vários experimentos em simulações verificou-se que, na maioria dos casos, ϵ e d são relativamente pequenos.

6.2 Protocolo de Drummond e Babaöglu

Todos os protocolos descritos até a seção anterior eram assíncronos, ou seja, não se baseavam na contagem do tempo físico para poderem avançar nas diversas etapas do protocolo. Nesta

seção descreveremos um protocolo síncrono que implementa difusão atômica para um grupo de comunicação. O sincronismo é aplicado neste protocolo através da limitação do tempo em que um processo pode se comunicar com os demais do grupo.

Rogério Drummond e Ozalp Babaöglu [DB84] mostram que selecionando a arquitetura de comunicação apropriada e replicando certos componentes de rede, é possível implementar uma difusão confiável em um sistema síncrono em apenas dois *estágios*, onde um estágio consiste de algum número arbitrário de passos de comunicação e computação com a restrição de que as mensagens enviadas por um processo não podem ser dependente das mensagens recebidas no mesmo estágio. O aspecto mais importante do modelo proposto, é que em outros protocolos síncronos tolerantes a t falhas, pelo menos $t+1$ estágios seriam necessários para qualquer solução de difusão confiável, mesmo com os mais restritivos modos de falhas.

6.2.1 Redes de Difusão Redundantes

A primitiva de comunicação fundamental implementada por uma rede de difusão é o *broadcast* (não-confiável). Usaremos a seguinte sintaxe para esta primitiva: *broadcast*(m); onde m é a mensagem sendo transmitida. Se nenhuma mensagem é recebida pelo protocolo de difusão dentro de um intervalo de tempo igual ao limite de tempo para uma transmissão na rede, então é dito que uma mensagem nula, denotada por ϕ, é recebida.

Apenas *broadcast*(m) não implementa uma difusão confiável pois uma rede de difusão efetivamente fornece conectividade-1[1] entre os processos. Para aumentar a conectividade, replicamos os componentes de comunicação por r. A rede resultante, chamada de *Rede de Difusão Redundante-R*, é mostrada na Figura 6.3. Este modelo requer r redes de difusão independentes onde cada um dos n processos está conectado a cada um dos r canais através das r ligações independentes.

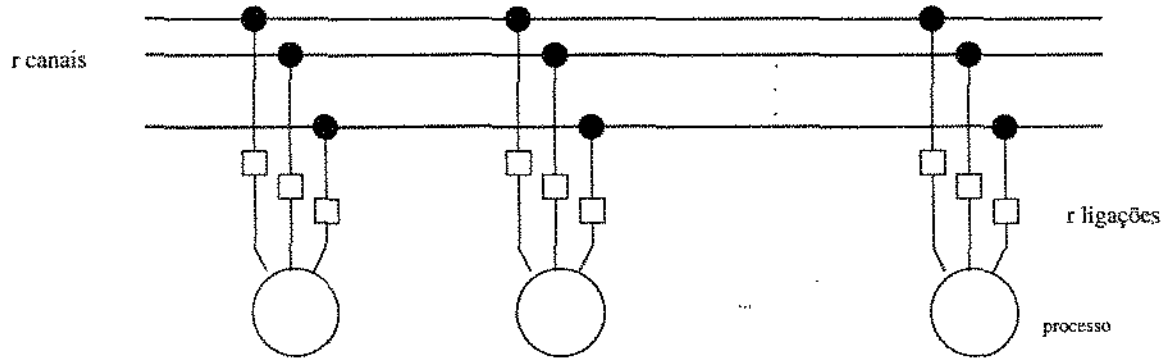


Figura 6.3: Rede de Difusão Redundante-R.

Seja π o número de falhas de processos, λ o número de falhas dos componentes de ligação, e γ o número de falhas dos canais de comunicação. A sintaxe da primitiva de comunicação é,

[1] A noção de conectividade está normalmente relacionada à contagem de caminhos distintos entre os nós de uma rede.

então, modificada para: $broadcast(m, i)$, onde, como antes, m é a mensagem a ser difundida, e $i \in \{1, 2, ..., r\}$ são os canais de comunicação pelos quais a difusão será realizada. $broadcast(m, *)$ significa que a difusão será feita em todos os canais do conjunto $\{1, 2, ..., r\}$.

Propriedade 1. *Em uma rede de difusão redundante-R, se $\lambda + \gamma < r$, então qualquer par de processos permanecerá pelo menos conectado-1 durante o protocolo. Ainda, não pode existir algum protocolo que implemente difusão confiável se $\lambda + \gamma \geq r$.*

A prova desta propriedade é dada em [DB84].

O Algoritmo

Seja $p_1, p_2, ..., p_n$ os processos no sistema de difusão. Sem perda de generalidade, assumimos que p_1 é o transmissor. Seja δ o valor original da mensagem que todos os processos não-falhos deverão concordar no caso do transmissor falhar. Seja V o domínio de todos os valores de mensagens possíveis não incluindo a mensagem nula ϕ.

Cada processo executa o protocolo apresentado no Algoritmo 6.2. Neste protocolo, C_r^i denota o conjunto de canais dos quais o processo p_i recebeu as mensagens (não-nulas) $m \in V$ no estágio r.

No estágio 1, o processo p_j difunde a mensagem em todos os canais de comunicação e decide pela própria mensagem (linhas 1 e 2). No segundo estágio, todos os processos p_i, para $i \neq j$, que receberam alguma mensagem m em algum dos canais, divulgam m em todos os canais nos quais não chegou nenhuma mensagem do estágio 1 (linhas 3 a 5). No fim do segundo estágio, os processos que ainda não decidiram por qualquer mensagem, decidem por m se esta foi recebida em algum dos canais, ou por δ caso nenhuma tenha sido recebida e, neste caso, todos decidem que o transmissor de m falhou.

6.2.2 Corretude

O seguinte teorema estabelece a correção deste protocolo.

Teorema: *Se $\lambda + \gamma < r$ e $n \geq \lambda + \pi$, então o protocolo implementa difusão confiável para processos que podem apresentar falhas de omissão.*

Prova. Note que, se $\lambda + \gamma < r$, então a Propriedade 1 assegura que quaisquer dois processos permanecem pelo menos conectados-1 durante a execução do protocolo. Procederemos pela análise de casos.

Se o transmissor é não-falho, todos os processos escutam a difusão no estágio 1 (e sabem que é de tal transmissor) e decidem em $m = v$ no estágio 2. Assim, a não-trivialidade é assegurada. O proposição seguinte assegura unanimidade no caso do transmissor falhar.

Se o transmissor é falho e não difunde nenhuma mensagem sobre os canais não-falhos nos quais ele está conectado no primeiro estágio, no fim do segundo estágio apenas mensagens ϕ foram recebidas e todos os processos não-falhos decidem por δ.

Difusão:

Estágio 1: Para p_j (o transmissor com o valor local $v \in V$)

broadcast$(v, *)$; 1

decida por v; 2

Recebimento:

Estágio 2: Para todo $p_i, i \neq j$

se $|C_1^i| \neq 0$ então 3

para cada k em $\{1, 2, ..., r\} - C_1^i$ faça 4

broadcast(m, k); 5

fim_para

decida por m; 6

fim_se

Entrega:

Na conclusão do estágio 2: Para todo p_i

se *não decidido ainda* então 7

se $|C_2^i| \neq 0$ então 8

decida por m; 9

senão

decida por δ; 10

fim_se

fim_se

Algoritmo 6.2: Protocolo de Drummond e Babaöglu.

Considere, agora, o caso onde o transmissor é falho e algum processo p_i não-falho escuta a difusão inicial. O protocolo requer que p_i divulgue v por todos os canais que não estão em C_1^i. Assim, no final do segundo estágio, as mensagens v terão alcançado todos os canais não-falhos aos quais p_i está conectado. Pela Propriedade 1, é garantido que todos os processos recebem v em um dos dois estágios. Como o recebimento de uma mensagem não-nula em um dos dois estágios implica em uma decisão, todos os processos concordam em v.

Por fim, consideraremos o que acontece se o transmissor é falho e apenas os outros processos escutam a difusão inicial. Isto pode ocorrer apenas quando todos os processos não-falhos têm ligações falhas para os canais não-falhos em que p_1 difundiu no estágio 1. O número mínimo de ligações falhas necessárias no fim do estágio 1 para este cenário é $n - \pi$ e isto acontece quando p_1 difunde usando um simples canal não-falho no qual ele está conectado e todos os processos não-falhos têm ligações falhas para este canal. Se $n \geq \lambda$ e $\lambda = n - \pi$, então não pode haver ligações falhas. Se um dos processos falhos ecoa v durante o estágio 2 em um canal diferente do que p_1 difundiu, então é garantido que todos os processos não-falhos escutam a mensagem e decidem por ela. Caso contrário, todos os processos não-falhos recebem apenas mensagens ϕ no final do estágio 2 e decidem por δ.

6.2.3 Desempenho

O protocolo visto gasta $n.r$ mensagens e requer dois estágios (*rounds*) de tempo pré-determinado (o sistema é síncrono) para implementar uma difusão confiável, onde n é o número de processos no grupo de comunicação e r é o número de canais redundantes de comunicação.

6.3 Comentários

A estratégia de propagação do protocolo de Molina e Spauster fornece uma alternativa viável para difusões ordenadas entre grupos. Entretanto, ela tem uma significante fraqueza: o custo substancial para se criar o grafo de propagação. Assim, essa abordagem apenas será de interesse se um número relativamente grande de mensagens é enviado durante o tempo de vida do grupo.

A importância do algoritmo de Drummond e Babaöglu é que ele mostra que determinadas arquiteturas podem ter propriedades particulares que podem ser usadas para avançar no projeto de novos sistemas distribuídos.

Capítulo 7

Uma proposta de protocolo baseado em ficha/anel e grafo de contexto

Como pôde ser observado no Capítulo 5, os protocolos que utilizam Grafo de Contexto para implementarem ordenação total, como o *Psync*, necessitam que todos os processos enviem uma mensagem a fim de determinarem que um conjunto de mensagens estáveis pode ser entregue. As outras abordagens com Grafo Causal, vistas nos protocolos *Total* e *ToTo*, também necessitam de um certo número de novas difusões para poderem entregar as mensagens estáveis. O fator principal de atraso destes protocolos, na entrega das mensagens, é que um processo só pode determinar que uma determinada mensagem de difusão tornou-se estável, e assim pode ser entregue em ordem total, quando ele tem certeza de que nenhuma outra mensagem enviada pelos outros processos tem alguma relação de causalidade com ela.

Neste capítulo é proposto um protocolo de difusão confiável que faz uma combinação das estruturas de Ficha/Anel, desenvolvidas por Chang e Maxemchuk, com a de grafo de contexto das mensagens de difusão, introduzida pelo Psync. A idéia é fazer com que os processos tenham condições de chegar a uma decisão mais rápido quanto ao estado global do sistema. Isto pode ser conseguido fazendo-se circular uma *ficha*, em um anel lógico contendo os processos do grupo, que informa aos processos quais foram as mensagens trocadas no grupo. Além disto, esta estratégia de difusão possibilita a detecção mais rápida de perda de mensagens e de processos que falharam.

O protocolo utiliza a ficha para informar aos processos qual é o grafo de contexto em concordância com o estado global do sistema. A cada passagem da ficha, um processo atualiza suas informações locais e repassa para o próximo do anel as novas informações de ordem das novas mensagens que ele recebeu.

7.1 Descrição do Protocolo

Assim como no protocolo de Chang e Maxemchuk, os processos pertencentes ao grupo são organizados em um anel lógico, e as difusões por *broadcast* de cada processo podem acontecer a qualquer instante. A ficha circula indefinidamente em sentido horário no anel, através de

mensagens ponto-a-ponto entre os processos vizinhos. Entretanto, diferentemente dos outros protocolos, a ficha carrega consigo uma estrutura de dados representando o grafo de contexto das mensagens difundidas no grupo.

Esta estratégia de difusão nos permitiu implementar dois serviços de entrega de mensagens:

- *Entrega Causal*: as mensagens são entregues obedecendo apenas as restrições de causalidade entre elas. Isto é, uma *mensagem causal* é entregue se todas as suas predecessoras na ordem causal já o foram.
- *Entrega Segura*: a mensagem é colocada em ordem total e, antes de sua entrega, é garantido que todos os processos já receberam esta mensagem.

O desenvolvedor da aplicação pode escolher um destes dois níveis de consistência entre mensagens quando necessitar de uma difusão para um grupo de processos. No nível seguro, as mensagens que estão em ordem causal e que já foram "vistas" por todos os processos podem ser postas em ordem total. A idéia é a mesma do protocolo Psync, através do conceito de *ondas*, onde uma onda contém aquelas mensagens que foram enviadas no mesmo tempo lógico, ou seja, são concorrentes. Os conjuntos de mensagens concorrentes (ondas) podem ser entregues em seqüência, formando uma ordem total.

As informações de causalidade fornecidas pelo grafo de contexto, e que são passadas entre os membros do grupo, permitem aos processos determinarem quais mensagens estão em ordem causal e quais mensagens tornaram-se *estáveis* e podem ser entregues em ordem total. Contudo, na nossa estratégia, não é necessário que todos os processos difundam alguma mensagem da aplicação para que uma outra torne-se estável. No caso de máximo atraso, basta apenas que a ficha dê uma volta no anel.

O grafo de contexto representa a relação de precedência das mensagens. Formalmente, seja M o conjunto das mensagens trocadas pelos processos do grupo de comunicação. Dadas duas mensagens $m, m' \in M$, se existe no grafo de contexto uma aresta direcionada ligando m' a m, então é porque a mensagem m' foi enviada no contexto da mensagem m, e assim, $m \rightarrow m'$. Note que o sentido da seta na relação *precede*, "$\rightarrow$", é invertido ao sentido das arestas no grafo.

Cada mensagem de difusão informa em seu cabeçalho a quais mensagens ela se liga no grafo de contexto, isto é, quais são suas predecessoras. Para que tenhamos sempre a redução transitiva do grafo [GU72], uma mensagem deve se ligar a apenas mensagens de uma mesma e mais recente *onda*. Uma onda no grafo contém apenas mensagens concorrentes, ou seja, mensagens que foram difundidas em um mesmo tempo lógico e, conseqüentemente, não possuem relação direta entre elas.

A Figura 7.1 mostra um exemplo de $G_{\rightarrow}$ e G para um grupo com quatro processos, A, B, C e D. b_1 e d_1 são as primeiras difusões de B e D, respectivamente, realizadas simultaneamente. a_1 foi enviada no contexto de b_1, e c_1 no contexto de b_1 e d_1. b_2 está no contexto de c_1 e, por transitividade, no contexto de b_1 e d_1.

A estratégia principal com este modelo é acelerar a entrega das mensagens. Os processos devem informar se a mensagem deve ser entregue em ordem causal ou em ordem total. Por exemplo, na Figura 7.1, dado que a mensagem b_2 requer o *serviço de entrega causal*, os processos não

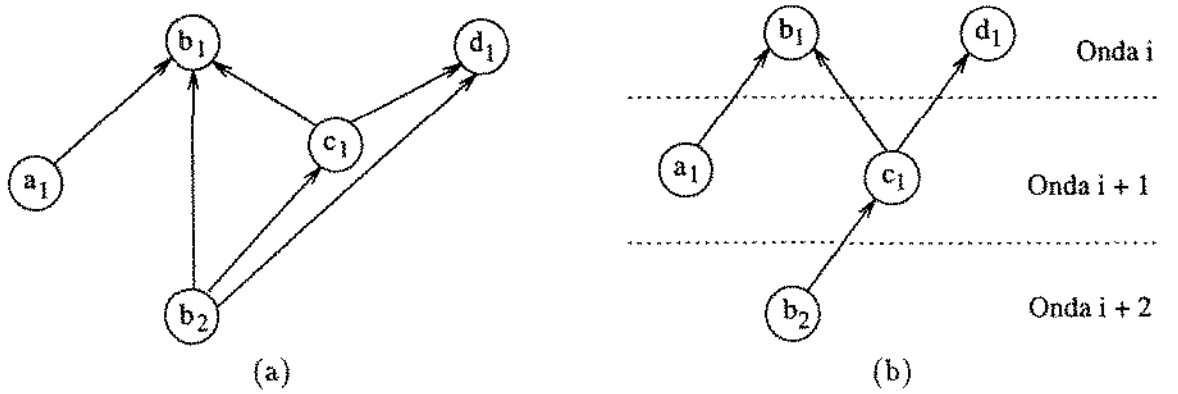


Figura 7.1: (a) $G_{\rightarrow}$ representa todas as relações de precedência entre as mensagens. (b) O grafo de contexto G é a redução transitiva de $G_{\rightarrow}$.

precisam esperar por a_1 para entregarem b_2. Para a entrega de b_2 é necessária, primeiramente, a entrega de c_1, que por sua vez, precisa de b_1 e d_1, e assim por diante. Para um processo que já recebeu e entregou b_1, d_1 e c_1, ele pode entregar b_2.

Agora, assuma que a_1 e c_1 requerem *entrega segura*. A entrega destas duas mensagens em ordem total, após a entrega das mensagens da onda i, pode ser feita assim que um processo percebe que não há nenhuma outra mensagem na onda $i+1$, ou seja, a_1 e c_1 tornaram-se estáveis. Isto pode ser determinado quando os outros processos, que não são os remetentes de a_1 e c_1, difundem alguma mensagem na onda seguinte, ou quando a ficha dá uma volta no anel e, conseqüentemente, todos os processo concordam com a entrega destas duas mensagens.

Estruturas de dados

O cabeçalho de cada mensagem de difusão m contém os seguintes campos:

- *m.tipo*: (CAUSAL | SEGURA). Informa o serviço de entrega requisitado para a mensagem.
- *m.precede*: Conjunto de identificadores das mensagens que precedem m.
- *m.msg_id*: Identificador da mensagem. É dividido em dois sub-campos:
  - *proc_id*: Identificador do remetente da mensagem.
  - *seq*: Número de seqüência (*timestamp*) da mensagem.

Se a mensagem for um pedido de retransmissão de alguma difusão, então o campo *msg_id* do cabeçalho da mensagem informa o identificador da mensagem requisitada.

Cada processo p_i do grupo mantém os seguintes dados:

- $onda_i$: Número da próxima onda para a entrega em ordem total, isto é, a onda corrente de processamento.
- seq_i: Número de seqüência da última mensagem difundida por p_i.

- GCL_i: Visão local de p_i do grafo de contexto da ficha (**GCF**).
- FMR_i: Fila de mensagens recebidas por p_i.

Cada vértice do grafo de contexto da ficha, GCF, representando as mensagens difundidas no grupo, contém, além das informações de cabeçalho de cada mensagem de difusão, um vetor de *bits* de tamanho n, $recebida[n]$, onde n é o número de processos no grupo. Cada elemento deste vetor representa um processo, e o valor do bit informa se o processo já recebeu a mensagem (1), ou não (0). Este vetor é iniciado com zeros.

7.1.1 Algoritmo

Os Algoritmos 7.1 e 7.2 mostram o pseudocódigo para o protocolo proposto. A função $causalidade(m, grafo)$, vista na linha 2, obtém o conjunto de mensagens da última onda que precedem na ordem parcial do $grafo$ uma mensagem m que se deseja difundir. Esta função mais o nível de entrega requerido para m (linha 1) determinam a "velocidade" de entrega de m. Se a função retorna todas as mensagens da última onda e m requer *entrega segura*, então a entrega poderá ser bastante retardada, já que m depende da entrega de todas as outras mensagens já difundidas. Por outro lado, se m requer *entrega causal* e a função retorna apenas mensagens de entrega causal, a entrega poderá ser feita assim que estas mensagens forem entregues. Assim, preferimos deixar a determinação da função *causalidade* em aberto, pois acreditamos que ela depende muito do tipo de aplicação que está se querendo desenvolver.

A descrição dos algoritmos é dada a seguir. Para enviar uma nova mensagem de difusão, o processo incrementa o seu contador de difusões de 1 e o atribui à nova mensagem (linhas 3 a 5). Antes de difundir a mensagem, o processo a insere em seu grafo de contexto local, GCL, e na fila de mensagens recebidas, FMR, (linhas 6 e 7). Se a mensagem requer *entrega causal* então a mensagem já pode ser entregue localmente (linha 8).

O processo p_i ao difundir uma mensagem m

determine o tipo de m, CAUSAL ou SEGURA, e atribua a $m.tipo$;	1
$m.precede := causalidade(m, GCL)$;	2
$seq := seq + 1$;	3
$m.msg\_id.proc\_id := p_i$;	4
$m.msg\_id.seq := seq$;	5
insira m em GCL_i e em FMR_i;	6
difunda m;	7
se ($m.tipo$ = CAUSAL) então	
entregue m;	8

Algoritmo 7.1: Fases de envio e *entrega causal* de uma mensagem.

Ao receber uma mensagem de difusão, o processo a insere em seu grafo de contexto local e na fila de mensagens já recebidas (linha 9). Em seguida, ele verifica se a mensagem é *causal* e se todas as suas predecessoras já foram entregues. Caso positivo, ele entrega a mensagem e todas

as outras que dependiam desta (linhas 10 e 11). Se a mensagem requer *entrega segura*, então o processo verifica se a *onda* corrente tornou-se estável. Formalmente, uma *onda* torna-se estável se:

$$\forall m \in onda, \forall x : 1..n \begin{cases} m.recebida[x] = 1 & \text{ou} \\ m.recebida[x] = 0 & \text{e} \quad p_x \text{ enviou } q, \text{ tal que } m \rightarrow q \end{cases}$$

Isto é, para toda mensagem da onda, todos os processos já receberam a mensagem após um giro da ficha no anel, ou existe alguma mensagem daqueles processos que ainda não atualizaram a ficha, tal que esta mensagem segue na ordem parcial a mensagem da onda sem confirmação de recebimento por tais processos. Isto demonstra que o processo pode reconhecer o recebimento de uma mensagem m tanto pelo vetor de bits *recebida* da ficha, quanto pelo envio de uma outra mensagem que segue m na ordem parcial. No segundo caso, o processo teria que já ter recebido m antes de enviar esta outra mensagem, pela própria definição da ordem causal.

No recebimento da ficha, o processo atualiza GCF inserindo as mensagens que ele recebeu enquanto a ficha circulava no anel e que não estão em GCF (linha 14). Em seguida, o processo percorre todo o grafo de contexto atualizando as o vetor *recebida* das mensagens que ele já recebeu, e requisitando a retransmissão daquelas mensagens que foram perdidas e ele deixou de receber (linhas 15 a 17). Novamente, a verificação da estabilidade da onda corrente é feita e desta vez, caso o resultado seja positivo, o processo retira de GCF a onda anterior à corrente, caso ela exista (linhas 18 a 20). Este procedimento evita que GCF aumente indefinidamente, retirando aquelas mensagens que já foram entregues por todos e não necessitam mais serem processadas. Por fim, o processo faz uma cópia de GCF em GCL e repassa a ficha com o grafo de contexto atualizado para o próximo do anel (linhas 21 e 22).

Finalmente, quando o processo recebe um pedido de retransmissão de uma mensagem m, se ele possui m ($m \in FMR$) então ele realiza uma nova difusão de m para o grupo.

7.2 Perda da Ficha e Detecção de Processos Falhos

Neste protocolo, assumimos que os processos podem falhar e que pode haver perdas de mensagens. Desta forma, o protocolo proposto, assim como outros protocolos baseados em ficha, está sujeito a um tipo especial de falha que pode ocorrer: *a perda da ficha*. Como a comunicação pode ser falha, a transmissão da ficha de um processo para outro no anel pode falhar, e como conseqüência disto o protocolo pode entrar em um estado de espera interminável. Um processo que falha gera um comportamento semelhante, já que conseqüentemente a ficha será perdida quando ela for enviada para tal processo. Nesta seção descrevemos que solução foi adotada para resolver este tipo de problema.

Considere que para a ficha dar uma volta no anel é gasto um período máximo de tempo T, e que este tempo será o mesmo para qualquer ponto do anel em que se inicia e termina o giro da ficha. Considere, também, que GCF e GCL contém um campo chamado $ficha\_id$ que representa a "versão" do grafo de contexto. Este campo é incrementado pelos processos antes de passar a ficha para o próximo do anel. Após o envio da ficha, o processo inicia um temporizador a fim de

O processo p_j ao receber uma mensagem de difusão m

insira m em GCL_j e em FMR_j; 9
se ($m.tipo$ = CAUSAL) então
  se ($\forall msg \in m.precede$, tal que msg já foi entregue) então
    entregue m; 10
    entregue as mensagens causais que seguem (e esperavam por) m; 11
  fim_se
fim_se
se a onda corrente tornou-se estável então
  entregue as mensagens seguras da onda corrente; 12
  $onda_j := onda_j + 1$; 13
fim_se

O processo p_j ao receber a ficha contendo GCF

para ($\forall msg \in GCL_j$, tal que $msg \notin GCF$) faça
  adicione msg a GCF; 14
fim_para;
para ($\forall msg \in GCF$) faça
  se ($msg \in FMR_j$) então
    $msg.recebida[p_j] := 1$; 15
  senão
    prepare uma mensagem ret com $ret.msg\_id := msg$; 16
    difunda ret; 17
  fim_se
fim_para
se a onda corrente tornou-se estável então
  retire de GCF a onda anterior; 18
  entregue as mensagens seguras da onda corrente; 19
  $onda := onda + 1$; 20
fim_se
copie GCF para GCL_j; 21
passe a ficha para o próximo do anel; 22

O processo p_j ao receber um pedido de retransmissão ret

se ($ret.msg\_id \in FMR_j$) então
  difunda $ret.msg\_id$; 23

Algoritmo 7.2: Fases de recebimento e *entrega segura* das mensagens.

medir o tempo em que a ficha dá uma volta no anel. Observe que isto cria um sistema pseudo-síncrono, já que limita o tempo de rotação da ficha no anel. Se o tempo T expirar, o processo p_i inicia a *Fase de Reforma* através de um *Algoritmo de Eleição*. Ele envia para o próximo do anel uma mensagem especial $elege_i$, que possui dois campos: $ficha\_id$, com o número de seqüência de GCL_i; e *ativos*, contendo o seu identificador p_i. Este último campo será atualizado por cada processo que receber a mensagem e, no final do algoritmo, conterá a lista de identificadores dos processos ativos.

Tal algoritmo elegerá o processo que tem a versão mais recente do grafo de contexto. Note que mais de um processo pode enviar tal mensagem, pois todos "esperam" pela ficha. Ao receber uma mensagem $elege_i$, todo processo p_j executa o Algoritmo 7.3, entrando também na fase de reforma.

O processo p_j ao receber $elege_i$
se ($elege_i.ficha\_id < GCL_j.ficha\_id$) então
adicione o identificador p_j a $elege_j$;
envie $elege_j$ para o próximo do anel;
senão
adicione o identificador p_j a $elege_i$;
envie $elege_i$ para o próximo do anel;
fim_se

Algoritmo 7.3: Fase de Reforma.

Ao receber $elege_i$, um outro processo p_j verifica se o $ficha\_id$ da mensagem é menor que o $ficha\_id$ do seu GCL. Caso positivo, p_j envia $elege_j$ com este $ficha\_id$ maior. Caso negativo, p_j repassa $elege_i$ para o próximo do anel. Antes de enviar uma mensagem *elege*, o processo adiciona seu identificador ao campo *ativos* de tal mensagem. Este procedimento se repete até que um processo p_k receba de volta $elege_k$, ganhando a "votação" que elege o processo que contém a versão mais recente da ficha. p_k envia então uma mensagem $eleito_k$ contendo a lista de processos ativos, retirada de $elege_k.ativos$. Ao receber esta mensagem, os outros processos saem da fase de reforma. Finalmente, p_k recomeça o protocolo enviando a nova ficha, extraído do seu GCL, para o próximo processo do novo anel.

7.3 Corretude

Para demonstrar que o protocolo é correto, precisamos provar as duas propriedades já discutidas nos outros capítulos.

Prova da propriedade de segurança

No protocolo proposto temos dois tipos de entrega: *causal* e *segura*. Assim, analisaremos os dois tipos separadamente:

- Entrega causal. Existem duas situações em que pode haver a entrega de uma mensagem causal.

  - Caso 1: Um processo p_i pode imediatamente entregar uma mensagem causal m_y que ele acabou de enviar (linha 8 do algoritmo) sem violar a ordenação causal.

    Para toda mensagem m_x que precede m_y, pela própria definição $m_x \rightarrow m_y$ implica que a entrega de m_x por p_i ocorreu antes do envio de m_y.

  - Caso 2: Um processo p_j só entrega uma mensagem causal m_y que ele acabou de receber (linha 10) quando todas as suas predecessoras já foram recebidas e entregues obedecendo, assim, a ordem causal.

    Se uma determinada mensagem m_y foi recebida por p_j, mas a sua predecessora $m_x \rightarrow m_y$ ainda não foi recebida, então p_j, ao receber a ficha, requisita a retransmissão de m_x e aguarda que ela seja recebida e entregue, antes de entregar m_y. Contudo, esta espera é limitada e a entrega de ambas as mensagens é assegurada pela propriedade de entrega.

- Entrega segura. A entrega de uma *mensagem segura* m_x por um processo p_j requer que ela seja entregue em ordem total e que todos os processos a tenham recebido.

  Esta entrega acontece sempre que a onda corrente torna-se estável. E isto pode ocorrer na recepção de uma mensagem de difusão ou da ficha (linhas 12 e 19). Nestas duas situações, o processo verifica o critério de decisão da equação 7.1.1, obedecendo a entrega segura.

Prova da propriedade de entrega

Toda mensagem m_x gerada pelo processo p_i é entregue pelo próprio processo p_i. O que devemos analisar é a entrega de m_x por um outro processo p_j. Para isto considere os três casos seguintes:

- Caso 1: p_i difunde m_x e todos os processos a recebem. Pelas situações de entrega discutidas acima, a propriedade é garantida.

- Caso 2: p_i difunde m_x e somente p_j não a recebe. Neste caso, p_j ao receber a ficha requisita a retransmissão de m_x (linhas 16 e 17), e qualquer processo p_k $(k \neq j)$ que receba este pedido, retransmite m_x.

  Assumindo que não há particionamento de rede, de forma que p_j fique isolado, e o sistema de comunicação é confiável a tal ponto de sempre poder entregar uma mensagem difundida, mesmo que ocorra um certo número de perda de mensagens, então podemos assumir que, em algum instante, o pedido de retransmissão de m_x é recebido por p_k e a sua retransmissão é recebida por p_j.

- Caso 3: p_i difunde m_x e nenhum outro processo a recebe. Idem ao caso anterior, onde p_j representa todos os processos exceto p_i, e $p_k = p_i$.

O caso 3 só será verdade se o processo p_i não falha logo após transmitir m_x. Uma solução para contornar este problema poderia ser a seguinte. Os processos, ao detectarem a falha de p_i e após um certo número de pedidos de retransmissão sem resposta, no final da Fase de Reforma, retiram do grafo de contexto a mensagem m_x, consolidando a sua não existência.

7.4 Desempenho do Protocolo

Para avaliar o desempenho do protocolo proposto, nós analisamos especificamente o custo em termos da latência para a entrega de uma mensagem difusão por cada processo. Consideraremos, assim como para os outros protocolos estudados, a situação em que não há qualquer tipo de falha durante a operação do protocolo.

O custo é dado pelo número de mensagens necessárias para a entrega de uma mensagem de difusão, incluindo a própria mensagem. Na entrega causal, dado uma mensagem m, se todas as suas predecessoras já foram recebidas e entregues, a sua entrega será imediata. Assim, o custo é 1, não considerando o custo para a ficha dar uma volta no anel, já que a entrega das mensagens causais não depende do giro da ficha.

Na entrega segura, o custo é, no pior caso onde há poucas difusões no grupo, n mensagens de passagem da ficha. No melhor caso, todos os processos geram difusões concorrentes em *ondas* e, assim, eles podem entregar as mensagens em ordem total sem esperar que a ficha circule no anel. Neste caso, o custo também é 1, já que após todas as difusões terem sido recebidas a onda corrente estará estável e pronta para a entrega. Note que este comportamento é semelhante ao do protocolo de Chang e Maxemchuk, onde há menos mensagens adicionais por cada difusão quando o sistema está ocupado do que quando está ocioso.

7.5 Comentários

O protocolo proposto parece ser uma alternativa viável aos protocolos discutidos nesta dissertação. Ao implementarmos ordenação causal das mensagens, nós conseguimos acelerar o processo de entrega das mensagens que não estão relacionadas entre si, o que é penalizado nos protocolos que utilizam anel/ficha e implementam ordenação total, como o Totem e o de Chang e Maxemchuk. Além disto, no Totem, o envio e a entrega de uma mensagem só é possível quando a ficha é recebida pelo processo. Isto pode levar um tempo considerável, dependendo da topologia e velocidade da rede.

Assim como o protocolo Psync, o protocolo proposto consegue implementar ordenação total a partir da ordem causal determinada pelo grafo de contexto. Contudo, no protocolo proposto, não é necessário que todos os processos difundam alguma mensagem para que uma outra possa ser entregue em ordem total. No máximo de latência de atraso, é necessário apenas que a ficha dê uma volta no anel.

Os protocolos Total e ToTo, apesar de requererem menos mensagens adicionais para a entrega de uma mensagem em ordem total, esta entrega não é *segura*, isto é, não requer que todos os

processos tenham recebido a mensagem antes de sua entrega. No Transis, isto é chamado de *entrega concordada*.

Ainda em comparação com os protocolos que utilizam grafo de contexto, a estratégia de solução proposta permite que se detecte, de forma mais rápida, processos que falharam e pararam de se comunicar no grupo, através do giro da ficha pelo anel.

Capítulo 8

Conclusão

O objetivo inicial do nosso trabalho era fazer um estudo detalhado do problema de se conseguir difusão confiável em grupos de comunicação, bem como dos principais protocolos publicados que procuram resolvê-lo. Em nossa opinião, esse objetivo foi realizado e tem como resultado os capítulos 2, 3, 4, 5 e 6 desta dissertação.

Acreditamos que proporcionamos ao leitor uma visão detalhada do problema de difusão confiável e dos protocolos desenvolvidos para a comunicação em grupo. Além disso, criamos uma classificação para a descrição dos protocolos estudados que acreditamos ser útil para estudos futuros.

O estudo dos protocolos de difusão também serviu de subsídio para a proposição de um novo protocolo, descrito no Capítulo 7, que é a segunda contribuição deste trabalho.

Este Capítulo está dividido em 3 seções. A seção 8.1 faz alguns comentários sobre os aspectos gerais dos protocolos de difusão confiável. A seção 8.2 analisa as diferentes abordagens desenvolvidas pelos protocolos para conseguir difusão confiável. A seção 8.3 encerra o texto com algumas sugestões para trabalhos futuros.

8.1 Comentários

Embora muitos protocolos de difusão confiável tenham sido desenvolvidos, muitos destes protocolos têm uma alta sobrecarga. Em TCP cada conversação ponto-a-ponto tem seu próprio mecanismo de reconhecimentos. Todos os protocolos de ordenação causal têm que manter as informações da ordem parcial. Estas sobrecargas, embora razoáveis para redes locais, tornam-se um fator dominante quando várias redes locais estão participando do protocolo.

Um problema que tem sido identificado pela comunidade de banco de dados, envolvida com aplicações distribuídas, é o particionamento de rede. Os protocolos que lidam com este problema de particionamento e junção de rede são aqueles que utilizam *Sincronismo Virtual Estendido*, como o Totem e o Transis. Os outros protocolos ou assumem que o particionamento não ocorre ou permitem somente que um componente primário continue, e não permitem a junção a menos que os processos se unam sem armazenamento estável intacto.

O número de mensagens por tempo de um protocolo de difusão confiável é seriamente degradado quando o número de retransmissões aumenta; as retransmissões requerem banda passante e tempo de processamento. Apesar disto, muitos protocolos permitem livre acesso ao meio de comunicação, e usam mecanismos de controle de fluxo externos. Estes mecanismos dependem de heurísticas para determinar a carga do tráfego corrente e são freqüentemente imprecisos, levando a uma sobrecarga dos buffers de entrada, perda de mensagens e degradação do desempenho. O sistema Totem foi projetado para especificamente lidar com este problema, e é o único que implementa o controle de fluxo no próprio protocolo de difusão.

8.2 Análise das soluções

A Tabela 8.1 a seguir sumariza todos os aspectos envolvidos em prover difusão confiável e o que cada protocolo analisado apresentou.

A estratégia de solução analisa as estruturas de dados e modelo de controle para se conseguir uma consistência nas difusões do grupo. A seguir vem os aspectos de ordenação das mensagens. Alguns dos protocolos apresentam mais de um modelo de ordenação, mas somente o Totem e o Transis abordam o modelo de ordenação segura, proveniente da utilização do mecanismo de sincronismo virtual estendido.

Os aspectos seguintes analisam a *Tolerância a Falhas* dos protocolos. A maioria deles permite falhas dos processos, mas poucos possibilitam a recuperação do processo sem ter que reconstruir a configuração do grupo. O particionamento e re-junção da rede enfoca a capacidade do protocolo suportar falhas no canal de comunicação, que resultam em uma divisão da rede em dois ou mais segmentos, sem que os processos que ainda se comunicam parem de operar.

Por fim, analisamos o desempenho de cada protocolo, como visto na Tabela 8.2. Este desempenho é medido para a maioria dos protocolos como o número de mensagens geradas por cada difusão; e para outros, onde, não há mensagens adicionais de controle, o tempo médio gasto para se chegar a uma concordância do grupo e entregar a mensagem. Este é o caso do protocolo Totem.

Na confecção desta tabela, foi considerado que os processos utilizam *broadcast* para se comunicarem e que não há perda de mensagens, ou seja, não se consideram retransmissões. Esta restrição na análise se deve ao fato de que nem todos os protocolos estudados tratam de falhas. Portanto, ressaltamos que as informações presentes nesta tabela não são a única ou a melhor forma de compará-los, visto que há vários outros parâmetros que podem influenciar na escolha de um protocolo. A tabela é apresentada apenas para efeito ilustrativo.

8.3 Sugestão para Trabalhos Futuros

Os capítulos 3, 4, 5 e 6 apresentam descrições de protocolos, tomando como base o modelo de sistemas proposto no Capítulo 2. São feitas demonstrações das operações destes protocolos e avaliações com relação ao número de mensagens enviadas e o tempo de execução. Contudo, não

	Chang e Maxemchuk	Totem	ISIS CBCAST	ISIS ABCAST	Raynal e Mostefaoui	Newtop	Psync	Trans/ Total	Transis	Molina e Spauster	Drummond e Babaõglu	Proposto
Estratégia de Solução	*Anel e Ficha*	*Anel e Ficha*	*Vetor de timestamps*	*Vetor de timestamps*	*Sincroniz. de Fases*	*Sincroniz. de Fases*	*Grafo de Contexto*	*Grafo de Contexto*	*Grafo de Contexto*	*Grafo de Propagacao*	*Redundancia de Rede*	*Anel/Ficha Grafo Cont.*
Ordenação Causal	*Não*	*Sim*	*Sim*	*Não*	*Sim*	*Não*	*Sim*	*Não*	*Sim*	*Não*	*Não*	*Sim*
Ord. Total (Concordada)	*Sim*	*Sim*	*Não*	*Sim*	*Não*	*Sim*	*Sim*	*Sim*	*Sim*	*Sim*	*Sim*	*Sim*
Ordenação Segura	*Não*	*Sim*	*Não*	*Não*	*Não*	*Sim*	*Não*	*Não*	*Sim*	*Não*	*Sim*	*Sim*
Ordenação entre Grupos	*Não*	*Sim*	*Sim*	*Sim*	*Sim*	*Sim*	*Não*	*Não*	*Não*	*Sim*	*Não*	*Não*
Sincronismo Virtual	*Não*	*Sim*	*Sim*	*Sim*	*Não*	*Não*	*Não*	*Não*	*Sim*	*Não*	*Não*	*Não*
Sinc. Virtual Estendido	*Não*	*Sim*	*Não*	*Não*	*Não*	*Não*	*Não*	*Não*	*Sim*	*Não*	*Não*	*Não*
Falhas de Processos	*Sim*	*Sim*	*Sim*	*Sim*	*Não*	*Sim*	*Sim*	*Sim*	*Sim*	*Não*	*Sim*	*Sim*
Recuperação de Processos	*Não*	*Sim*	*Não*	*Não*	*Não*	*Não*	*Não*	*Sim*	*Sim*	*Não*	*Sim*	*Não*
Particionam. de Rede	*Não*	*Sim*	*Não*	*Não*	*Não*	*Sim*	*Não*	*Não*	*Não*	*Não*	*Sim*	*Não*
Re-junção de Rede	*Não*	*Sim*	*Não*	*Não*	*Não*	*Não*	*Não*	*Não*	*Não*	*Não*	*Sim*	*Não*

Tabela 8.1: Análise comparativa dos protocolos

Protocolos	Medida	Desempenho	Observações
Chang e Maxemchuck	Mensagens por difusão	$L+1$	L é um parâmetro que determina o grau de resistência à falhas do sistema.
Totem	Tempo médio de difusão	$n.b + n.m.r.a$	Ver seção 3.2.2 para a descrição das variáveis.
ISIS CBCAST	Mensagens por difusão	1	Assumindo que as mensagens CBCAST precedentes já foram recebidas.
ISIS ABCAST	Mensagens por difusão	$1 + 1/k$	Uma mensagem **sets-order** é enviada depois de k ABCAST's.
Raynal e Mostefaoui	Mensagens por difusão	n	Ordenação causal entre grupos.
Newtop	Mensagens por difusão	n	Ordenação total entre grupos.
Psync	Mensagens por difusão	1 n	Ordenação causal. Ordenação total.
Trans/ Total	Mensagens por difusão	$(n+k+1)/2$	$k < n/3$ é o grau de resistência do sistema.
Transis	Mensagens por difusão	$n/2$ n	Entrega concordada. Entrega segura.
Molina e Spauster	Mensagens por difusão	$d+1$	d é a profundidade da árvore de propagação de mensagens entre grupos.
Drummond e Babaöglu	Mensagens por difusão	$n.r$	r é o número de canais de comunicação.
Proposto	Mensagens por difusão	1 n	Entrega causal. Idem ao CBCAST. Entrega segura. n mensagens ponto-a-ponto para passar a ficha no anel.

Tabela 8.2: Desempenho dos protocolos

é feito um estudo mais apurado sobre a capacidade dos protocolos reagirem a falhas no sistema. Como um trabalho futuro, sugerimos uma análise dos algoritmos para gerenciamento do grupo na ocorrência de falhas, como parada de processos e particionamento e junção de rede.

Seria interessante que, em um trabalho futuro, o protocolo proposto no Capítulo 7 fosse implementado a fim de melhor analisá-lo em termos práticos. Esta implementação seria importante para se obter uma avaliação realística da robustez e eficiência do protocolo proposto e também para possibilitar comparações mais efetivas com outros protocolos já implementados.

Bibliografia

[AC93] Y. Amir; L. E. Moser; P. M. Meliar-Smith; D. A. Agarwal and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the IEEE 13th International Conference on Distributed Computing Systems*, pages 551–560, Pittsburgh, PA, May 1993.

[AC95] Y. Amir; L. E. Moser; P. M. Meliar-Smith; D. A. Agarwal and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, November 1995.

[Aga94] Deborah A. Agarwal. *Totem: A Reliable Ordered Delivery Protocol.* PhD thesis, University of California, 1994.

[Ami95] Yair Amir. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, Hebrew University of Jerusalem, 1995.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[BJ87] K. P. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5:47–76, February 1987.

[BLP96] L. E. Moser; P. M. Meliar-Smith; D. A. Agarwal; R. K. Budhia and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant group communication system. Technical report, Department of Electrical and Computer University of California, Santa Barbara, CA 93106, 1996.

[BM96] R. Renesse; K. P. Birman and Silvano Maffeis. Horus: A flexible group communication system. Technical report, Dept. of Computer Science, Cornell University, 1996. Submitted for publication in the Communication of The ACM.

[CM84] J. M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):39–59, August 1984.

[DB84] R. Drummond and O. Babaoglu. Streets of bizantium: Network architectures for fast reliable broadcasts. Technical Report TR 84-613, Department of Computer Science, Cornell University, New York, 1984.

[DK94] G. Collouris; J. Dollimore and T. Kindberg. *Distributed Systems.* Addinson-Wesley, second edition, 1994.

[ES93] R. A. Macedo; P. D. Exhilchelvan and S. K. Shrivastava. Newtop: A total order multicast using causal blocks. Technical report, Dept. of Computing Science, University of Newcastle, Newcastle upon Tyne, 1993.

[ES94] R. A. Macedo; P. D. Exhilchelvan and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. Technical report, Dept. of Computing Science, University of Newcastle, Newcastle upon Tyne, 1994.

[GMS91] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9:242–271, August 1991.

[GU72] A. Aho; M. Garey and J. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, pages 131–137, 1972.

[HB89] M. F. Kaashoek; A. S. Tanembaum; S. Hummel and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23:5–19, October 1989.

[HK95] R. Renesse; K. P. Birman; R. Friedman; M. Hayden and D. A. Karr. A framework for protocol composition in horus. In *Proc. of the Fourteenth ACM Symp on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, August 1995. ACM SIGOPS-SIGACT.

[Jal94] P. Jalote. *Fault tolerance in distributed systems*, chapter Reliable, Atomic and Causal Broadcast. Prentice-Hall, 1994.

[Kra93] Shlomo Kramer. Total ordering of messages in multicast communication systems. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, 1993.

[KT91] M. F. Kaashoek and A. S. Tanembaum. Group communication in the amoeba distributed system. In IEEE, editor, *Proceding of 11th International Conference on Distributed Computing Systems*, pages 222–230, 1991.

[KT92] M. F. Kaashoek and A. S. Tanembaum. *Efficient Reliable Group Communication for Distributed Systems.* PhD thesis, The Vrije Universiteit, 1992.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LEMA94a] P. M. Melliar-Smith L. E. Moser and V. Agrawala. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):459–473, May 1994.

[LEMA94b] P. M. Melliar-Smith L. E. Moser, Y. Amir and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, June 1994. IEEE.

[LLPS89] N. C. Buchholz L. L. Peterson and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[Mal94] D. Malki. *Multicast Communication for High Availability*. PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.

[MR93] A. Mostefaoui and M. Raynal. Causal multicast in overlapping groups: Towards a low cost approach. In *IEEE Int. Conf. on Future Trends of Dist. Comp. Systems*, Lisboa, Portugal, September 1993.

[PMMSA90] L. E. Moser P. M. Melliar-Smith and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1:17–25, January 1990.

[RB91] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 341–353, Montreal, Quebec, Canada, August 1991.

[RM89] B. Rajagopalan and P. K. McKinley. A token-based protocol for reliable ordered multicast communication. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 84–93, Seatle, WA, October 1989.

[Ros93] Thierson Couto Rosa. Validação de ações atômicas distribuídas. Master's thesis, Departamento de Ciência da Computação, Universidade Estadual de Campinas, 1993.

[RS96] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *IEEE Computer*, February 1996.

[Sch88] Frank Bernhard Schmuck. *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems*. PhD thesis, Department of Computer Science, Cornell University, 1988.

[SMS91] L. L. Peterson S. Mishra and R. D. Schlichting. A membership protocol based on partial order. In *Proceedings on the International Working Conference on Dependable Computing for Critical Applications*, volume 6, pages 309–331, Tucson, AZ, February 1991.

[SS91] K. P. Birman; A. Schiper and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9:272–314, August 1991.

[Ste94] W. R. Stevens. *TCP/IP Illustrated, Volume 1 - The Protocols*, chapter Broadcasting and Multicasting. Addinson-Wesley, 1994.

[YAM92a] Shlomo Kramer Yair Amir, Danny Dolev and Dalia Malki. Membership algorithms for multicast communication groups. In *Proceedings of the 6th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 647*, pages 292–312, Haifa, Israel, November 1992.

[YAM92b] Shlomo Kramer Yair Amir, Danny Dolev and Dalia Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, MA, July 1992.