Universidade do Minho
Departamento de Informática

Comunicação Fiável em Sistemas Distribuídos em Grande Escala [†]

por

José Orlando Roque Nascimento Pereira

Dissertação apresentada à Universidade do Minho para obtenção do grau de Mestre em Informática, na especialidade de Sistemas Distribuídos, Comunicações por Computador e Arquitectura de Computadores

Orientador:
Francisco Soares de Moura
(Professor Associado)

Braga
Dezembro de 1997

[†]Trabalho apoiado pela JNICT, através do Programa PRAXIS XXI, ref. BM/6663/95

Agradecimentos

Ao meu orientador, Prof. Francisco Soares de Moura, pelo seu apoio e pelas suas sugestões e críticas que contribuíram para melhorar este trabalho.

Ao Rui Oliveira, pela sugestão dos sistemas distribuídos tolerantes a faltas como uma interessante àrea de trabalho, pelas longas discussões das ideias fundamentais desta tese e pelo apoio constante ao longo deste ano de trabalho.

A todos os restantes membros do Grupo de Sistemas Distribuídos, pelo incentivo constante, pelo ambiente de trabalho e acima de tudo pelas numerosas e frutíferas discussões.

Aos que tiveram a bondade de ler e comentar as nem sempre muito estimulantes versões preliminares deste texto.

Resumo

Protocolos de difusão fiável em grupos de processos são uma ferramenta de programação utilizada em sistemas distribuídos e confiáveis para isolar cada aplicação da complexidade decorrente da comunicação e das diversas faltas que podem ocorrer no sistema. Isto é conseguido oferecendo às aplicações uma abstração de um canal de difusão que oferece garantias muito fortes quanto à qualidade de serviço, o que é suficiente para garantir a correcção de um largo espectro de aplicações com um esforço mínimo por parte do programador.

A concretização de protocolos de difusão fiável sobre redes em grande escala introduz um nível adicional de complexidade, pois as características destas redes dificultam as concretização correcta e eficiente de serviços que oferecem garantias muito fortes.

Diversas propostas procuram ultrapassar este problema disponibilizando serviços que oferecem menos garantias de qualidade de serviço, mas que apresentam desempenho superiror nas situações em que são utilizáveis. No entanto, quanto menos garantias se oferecerem, menos aplicações podem ser correctamente suportadas, uma vez que diferentes aplicações toleram diferentes relaxamentos da qualidade de serviço.

Nesta tese propõe-se um protocolo de difusão fiável que possa ser configurado para cada aplicação, de modo a poder ser simultâneamente eficiente e correcto porque adequado à aplicação em causa.

Para o efeito, começa-se por decompor a especificação de protocolos de difusão segundo um conjunto de parâmetros aplicáveis em separado a cada uma das mensagens de uma sessão, que podem ser combinados em especificações de protocolos adequados a cada aplicação.

Propõe-se então uma estratégia de concretização aberta orientada por objectos, que permite compôr protocolos a partir de módulos independentes para cada mensagem, correspondentes aos parâmetros de especificação.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Nos últimos anos assistiu-se a nível global a um aumento exponencial do número de utilizadores de redes públicas de comunicação de dados. Este crescimento foi incentivado pela popularidade crescente quer do correio electrónico, quer dos serviços de publicação e consulta de informação, como a USENET e sobretudo a WORLD WIDE WEB.

Em resposta à procura desencadeada por esse número crescente de utilizadores de redes públicas de comunicação de dados, assistiu-se também a um desenvolvimento acelerado, tanto em qualidade com em quantidade, das infra-estruturas de comunicação usadas em redes em grande escala.

Como consequência, torna-se atraente a utilização destas infra-estruturas para outras aplicações que não a troca de correio electrónico e a publicação de informação. Nomeadamente, têm sido constituídas com sucesso redes em grande escala que inter-ligam as redes locais de organizações geograficamente dispersas.

Passa então a ser desejável o alargamento das aplicações existentes de redes locais a redes geograficamente distribuídas. Algumas destas aplicações coincidem já com aplicações típicas de redes em grande escala, como por exemplo, o correio electrónico, podendo imediatamente ser aplicada a tecnologia existente.

No entanto, para outras aplicações, a tecnologia disponível em redes em grande escala revela-se inadequada para ser reutilizada nas aplicações típicas em redes locais. É este o caso das aplicações que envolvem a replicação de bases de dados para tolerância a faltas e obtenção de melhor desempenho. Isto acontece porque as tecnologias de replicação desenvolvidas para redes em grande escala não proporcionam suficientes garantias de coerência da informação na presença de actualizações frequentes e concorrentes, destinando-se apenas à distribuição de informação imutável ou onde alguma incoerência é tolerada pelos utilizadores.

Por outro lado, a tentativa de reutilização de soluções desenvolvidas estritamente no contexto de redes locais, enfrenta os novos obstáculos decorrentes das características das redes em grande escala. Apesar de serem funcionalmente idênticas a redes locais, são fundamentalmente diferentes em termos de faltas a que estão sujeitas e em termos do desempenho que pode ser obtido. Nomeadamente, passa a ser necessário considerar as características das inter-ligações, que introduzem a possibilidade de partições da rede e a imprevisibilidade de atrasos e de largura de banda disponível, bem como o impacto do crescimento do número de nós da rede na complexidade dos algoritmos utilizados. Outra característica que distingue as redes em grande escala de redes locais é a diversidade dos seus componentes, desde a capacidade de cálculo ao fabricante, passando pela entidade que administra cada um deles.

Ao mesmo tempo, começa também a ser comum a utilização de equipamentos de computação nómada, que possuem capacidades de cálculo e comunicação diversas mas limitadas, o que faz com que redes locais onde existam esses equipamentos exibam também algumas das propriedades típicas de redes em grande escala.

Convém então, por um lado, analisar porque é que as características das redes em grande escala representam um problema para a reutilização de soluções desenvolvidas em redes locais. Por outro lado, é conveniente investigar as razões do sucesso das aplicações tradicionais de replicação de informação existentes em sistemas distribuídos em grande escala.

Em primeiro lugar, exemplos típicos de aplicações tolerantes a faltas comuns em redes locais são baseadas em protocolos de difusão fiável totalmente ordenada ou em transacções atómicas. Ambas as abordagens, de modo a serem soluções genéricas, procuram assegurar a correcção de um largo espectro de aplicações com um esforço mínimo por parte dos programadores. Isto é possível pois ambas proporcionam um conjunto de garantias extremamente fortes quanto à comunicação e detecção de faltas, que são questões chave da programação de sistemas distribuídos. Acontece que, para disponibilizar essas garantias fortes, são assumidos alguns pressupostos que nas redes em grande escala existentes se revelam, na melhor das hipóteses, bastante optimistas.

Por outro lado, as soluções existentes em redes em grande escala tendem a oferecer muito poucas garantias quanto à qualidade de serviço, limitando-se geralmente a assegurar o maior esforço. Ao mesmo tempo, caracterizam-se pela predominância de soluções específicas para cada problema, de modo a tirar vantagem da semântica da informação replicada. Por exemplo, existem soluções distintas e independentes para a replicação de ficheiros, de notícias e de hipertexto.

Esta dificuldade de reutilização das soluções previamente desenvolvidas em qualquer dos contextos é especialmente grave na medida em que as características típicas de uma rede em grande escala tornam ainda mais necessária

a construção de sistemas de replicação de informação, tanto para tolerância de faltas como para melhoria do desempenho.

1.2 Objectivos

Da avaliação dos problemas levantados pelo desenvolvimento de aplicações distribuídas e confiáveis em sistemas em grande escala, bem como das propostas de solução existentes, especialmente no que diz respeito a replicação de serviços, resultam os dois pressupostos em que se fundamenta esta tese:

- o bom desempenho de uma aplicação num sistema distribuído em grande escala requer que o sub-sistema de comunicação usado não obrigue a aplicação a utilizar mais garantias do que as necessárias;
- como a correcção de diferentes aplicações depende de diferentes garantias por parte do sub-sistema de comunicação, é necessário que esse sistema possa ser configurado para tirar partido da semântica do problema, sendo as tradicionais classes de qualidade de serviço insuficientes.

Em resposta, o objectivo desta tese é procurar uma estratégia que permita concretizar protocolos de comunicação adaptáveis às necessidades de diferentes aplicações. Concretamente, procura-se realizar este objectivo com o desenvolvimento de:

- uma especificação modular de serviços de difusão fiável em grupos de processos, capaz de descrever as necessidades de um largo espectro de aplicações;
- uma arquitectura e um conjunto de infra-estruturas de *software* para concretização de especificações modulares de protocolos configuráveis.

Apresenta-se então, como demonstração da viabilidade da estratégia de concretização de protocolos escolhida, um conjunto de componentes que podem ser combinados para formar protocolos de difusão fiável adaptáveis à semântica de aplicações.

1.3 Estrutura da tese

O Capítulo 2 introduz os conceitos de confiabilidade e tolerância a faltas, apresentando a redundância como uma estratégia genérica para obtenção de sistemas tolerantes a faltas. Discutem-se quais os problemas genéricos na concretização de soluções de replicação dando particular ênfase ao paradigma da difusão fiável em grupos de processos como uma ferramenta adequada à concretização de serviços replicados.

No Capítulo 3 examina-se em detalhe um sistema distribuído em grande escala, com base nas suas características e nos requisitos das suas aplicações, apresentando-se o paradigma da difusão fiável em grupos de processos como uma ferramenta desejável para a sua programação. São descritas neste capítulo diversas tentativas de solução dos problemas encontrados em sistemas distribuídos em grande escala, das quais se pode identificar a ideia comum de relaxar as garantias oferecidas à aplicação para obter um desempenho superior.

O Capítulo 4 apresenta uma revisão de diversas arquitecturas e infra-estruturas de suporte à concretização de protocolos, incluindo a estratificação em camadas, programação por objectos e redes activas, sendo comparadas em termos da possibilidade de configuração dos protocolos resultantes por parte dos programadores de aplicações.

O Capítulo 5 introduz o conceito de protocolos de comunicação configuráveis como o paradigma adequado à programação de sistemas distribuídos em grande escala. Para o efeito é apresentado um modelo detalhado para um sistema distribuído, sobre o qual se define precisamente o que se entende por protocolo de comunicação. Define-se então o que é um protocolo de comunicação configurável e quais os aspectos relevantes para os especificar e concretizar. Finalmente, apresenta-se uma especificação modular para protocolos de difusão fiável, adequada a ser utilizada em protocolos configuráveis.

O Capítulo 6 apresenta uma estratégia para concretização de protocolos configuráveis. Em primeiro lugar apresenta-se uma infra-estrutura de componentes que define como podem ser concretizados e combinados módulos de *software*. Em seguida, uma infra-estrutura de protocolos que suporta a concretização aberta orientada por objectos de protocolos de comunicação, permitindo às aplicações configurar um protocolo com base na escolha de meta-objectos para cada mensagem. Finalmente, apresenta-se uma infra-estrutura de representação de histórias de processo como elemento chave para a inter-operação entre diferentes módulos.

O Capítulo 7 demonstra a concretização aberta e orientada por objectos com base nas infra-estruturas do Capítulo 6, de protocolos de difusão cumprindo as especificações do Capítulo 5.

Finalmente o Capítulo 8 resume os resultados e conclusões, propondo-se então possíveis linhas de investigação que podem ser exploradas na sequência desta tese.

Capítulo 2

Confiabilidade em sistemas distribuídos

2.1 Motivação

Um sistema de computação distribuída é constituído por um conjunto relativamente vasto de componentes tanto de *hardware* como de *software* que naturalmente estão sujeitos a falhar. Em muitos sistemas, a falta de um componente provoca a indisponibilidade do serviço prestado, podendo inclusivé induzir a modos de funcionamento lesivos para outros componentes e sistemas. Outros sistemas estão projectados para tolerarem faltas, mascarando-as de modo a que o serviço não seja interrompido, ou na pior das hipóteses, interrompendo-o imediatamente sem causar mais danos.

Apesar de em algumas utilizações a indisponibilidade ser aceite pelos utilizadores, existe um número crescente de aplicações de sistemas de computação em que o custo de uma interrupção do serviço é inaceitável, pondo em perigo bens materiais consideráveis ou mesmo vidas humanas.

Como consequência, surge a necessidade de aferir o desempenho de sistemas de computação distribuída segundo medidas de confiabilidade e de os projectar de acordo com estratégias que lhe confiram os comportamentos desejados na presença de faltas.

A aplicação das estratégias genéricas de confiabilidade a um sistema de computação distribuído resultam na introdução de redundância em alguns dos seus componentes. Embora existam soluções baseadas em *hardware* redundante, são especialmente interessantes do ponto de vista económico as soluções baseadas apenas em *software*.

A concretização de qualquer sistema com serviços redundantes implica porém a manutenção da coerência das diversas cópias existentes. Isto faz dos mecanismos destinados a assegurar essa coerência, uma questão fundamental na programação de sistemas distribuídos tolerantes a faltas.

2.2 Confiabilidade

2.2.1 Definição

Uma medida de *confiabilidade* de um sistema é uma avaliação da qualidade do serviço prestado por um sistema durante um período de tempo relativamente longo. A qualidade de serviço pode ser avaliada segundo vários critérios [KV93]:

Fiabilidade: Medida de disponibilidade contínua de um serviço, definida como a probabilidade do sistema funcionar conforme especificado durante um dado intervalo contínuo de tempo.

Segurança: Medida da capacidade de evitar falhas catastróficas, definida em dois sentidos:

i. probabilidade de o sistema não falhar de um modo catastrófico para o ambiente durante um dado intervalo de tempo;

ii. capacidade de evitar uma falha catastrófica para o sistema induzida pelo ambiente.

Disponibilidade: Medida da disponibilidade de um sistema em termos de alternância entre os estados de disponibilidade e indisponibilidade, definida como a probabilidade de num instante o sistema estar disponível.

Reparabilidade: Medida do tempo necessário para reparar um sistema depois de uma falha, definida como a probabilidade de o sistema já estar reparado algum tempo depois da falha.

Para cada sistema, a definição de confiabilidade depende das expectativas dos utilizadores em termos de qualidade de serviço.

2.2.2 Tolerância a faltas

No sentido de atingir uma meta em termos de confiabilidade, é necessária a avaliação objectiva das possibilidades de falta de cada um dos vários componentes do sistema e o estudo das relações entre eles em termos de confiabilidade [Cri91].

Para o efeito, um sistema de computação distribuída pode ser definido como um conjunto de *servidores* que prestam determinados *serviços*. Cada servidor pode ser, por sua vez, cliente de outros serviços, pelo que o correcto funcionamento do sistema como um todo está relacionado com o correcto funcionamento dos diversos servidores dos quais depende.

As *dependências* entre servidores são normalmente analisadas em termos funcionais. No entanto, neste contexto importa sobretudo analisar as dependências em termos de confiabilidade, no sentido em que as medidas de

confiabilidade de um servidor são influenciadas pelas medidas de confiabilidade de um outro.

Deste modo, a *falha* de um sistema, sendo um desvio da especificação do serviço prestado por um servidor, é a exposição de um estado interno incorrecto, ou *erro*. Pode então identificar-se como causa do erro a *falta* de um serviço que lhe deu origem [KV93].

É pois possível a ocorrência de uma falta e consequente erro sem que ocorra uma falha, o que acontece se, apesar do erro, o serviço prestado pelo sistema em questão continuar a obedecer à especificação, ilustrando a possibilidade de existência de sistemas *tolerantes a faltas*.

2.2.3 Estratégias para confiabilidade

O objectivo fundamental do projecto de um sistema tolerante a faltas é detectar e mascarar ou reparar os erros antes que estes apareçam como uma falha do sistema. Para o efeito, o fundamento de qualquer estratégia de tolerância de faltas é a introdução de *redundância* [KV93].

Quando um servidor depende de um outro serviço, a introdução de redundância significa que existem vários servidores a prestar-lhe esse mesmo serviço. Sendo esses servidores independentes no que diz respeito às falhas, nas situações em que há a possibilidade de a partir de um conjunto de respostas extrair a resposta correcta, apesar de algumas delas poderem não ser obtidas ou estarem erradas, é possível assegurar que os eventuais erros não se propagam como falhas do sistema.

Por exemplo, para evitar a falha de um computador é usada redundância física, utilizando vários discos para os quais são efectuadas em simultâneo todas as escritas. Em sistemas de comunicação é usada redundância no tempo, retransmitindo a mesma informação repetidamente para mascarar faltas temporárias do canal de comunicação. Outra técnica bem conhecida é a redundância de informação, que consiste na utilização de somas de verificação e dígitos de paridade para detectar e eventualmente corrigir faltas em qualquer mecanismo de armazenamento e transmissão de informação.

A redundância necessária e os métodos concretos usados para detectar e mascarar erros dependem do tipo de faltas que se pretende tolerar e do funcionamento do sistema. A introdução de redundância implica no entanto custos acrescidos, pelo que é necessário estabelecer um compromisso entre os custos decorrentes da falha do sistema e os custos de mecanismos de tolerância a faltas.

2.2.4 Caracterização de faltas e falhas

Tanto as falhas como as faltas podem ser caracterizadas em termos de um conjunto de parâmetros de modo a definir estratégias apropriadas [Lap92]. Destas há a salientar a distinção entre falhas benignas, que são as falhas por

paragem e omissão, das falhas malignas, ou seja, arbitrárias, afirmativas ou bizantinas, que conduzem a estratégias diferenciadas de tolerância.

A composição hierárquica de sistemas introduz um nível adicional de complexidade. Por exemplo, uma falha por paragem de um relógio pode induzir a comportamento arbitrário de sistema que depende de um relógio estritamente crescente para etiquetar eventos [Cri91]. Este tipo de fenómenos, em que uma falha possivelmente benigna de um servidor conduz à falha maligna de outro servidor é conhecido como *amplificação de falhas*.

2.3 Sistemas de computação distribuída

2.3.1 Modelo genérico

Como modelo para um sistema de computação distribuída[1], considera-se um conjunto de processos sequenciais e independentes, interligados por canais de comunicação. Toda a comunicação entre processos é efectuada por passagem de mensagens pelos canais.

Cada processo é uma máquina de estados, sendo a execução de cada processo dada pela sua história, que se define como uma sequência das suas transições de estado, ou passos, que podem ser emissão de mensagens, recepção de mensagens ou computação local.

A completa caracterização deste modelo depende ainda do sincronismo e das faltas assumidas, tanto dos canais como dos processos.

2.3.2 Sincronismo

Um modelo síncrono assume que existem limites bem conhecidos para a velocidade relativa entre processos e para o tempo de comunicação, o que significa que existe um tempo global que pode ser conhecido com alguma precisão.

Num modelo assíncrono, a inexistência de um referencial global de tempo significa que não podem ser estabelecidos quaisquer limites tanto para a velocidade relativa dos processos, como para o tempo de comunicação.

É ainda possível considerar modelos intermédios, em que há garantias mais fracas [DLS88] ou então apenas probabilísticas [HB96] quanto à exactidão dos relógios. Porém, o modelo assíncrono é o mais genérico de todos, uma vez que engloba os sistemas síncronos e semi-síncronos na medida em que continua neles correcto qualquer algoritmo que tenha como pressuposto apenas um sistema assíncrono.

Isto faz com que seja o modelo assíncrono o mais pessimista em relação aos pressupostos que são feitos sobre um sistema real, e como tal o mais apropriado a estabelecer limites inferiores e demonstrações de possibilidade.

[1] Ou neste contexto, apenas um *sistema distribuído*.

Em contrapartida, não é possível com um modelo assíncrono raciocinar sobre propriedades temporais de sistemas, o que o torna inadequado para outras situações.

2.3.3 Falhas de processos

As falhas de um processo estão relacionadas quer com transições de estado, quer com o envio e recepção de mensagens [HT93]. No que diz respeito a transições, um processo pode falhar por paragem ou então efectuando transições arbitrárias de estado.

Um processo pode ainda falhar tanto na recepção como no envio de mensagens, por omissão perdendo mensagens, ou então afirmativamente, recebendo e enviando mensagens arbitrárias.

2.3.4 Falhas de canais

Um canal de comunicação pode falhar por omissão, perdendo mensagens, ou afirmativamente, duplicando ou criando mensagens arbitrárias [HT93].

A perda pode ainda ser quantificada e relacionada com a possibilidade de falha por parte dos processos, uma vez que não faz sentido falar, por exemplo, da recepção de uma mensagem por um processo que falhou por paragem. Deste modo define-se canal sem perdas como um canal em que qualquer mensagem enviada a um processo que a tenta receber indefinidamente, é inevitavelmente recebida.

Em termos de perdas de mensagens convém distinguir três modelos distintos [BCBT96]:

Perda finita: Se um processo tenta receber indefinidamente um número infinito de mensagens que lhe foram enviadas, então o canal não lhe entrega apenas um subconjunto finito dessas mensagens.

Perda justa: Se um processo tenta receber indefinidamente um número infinito de mensagens que lhe foram enviadas, então o canal entrega-lhe um subconjunto infinito dessas mensagens.

Perda infinita: O canal entrega apenas um sub-conjunto finito das mensagens enviadas.

É de salientar ainda que o último caso se trata de uma falha permanente e não temporária, o que implica a utilização de estratégias fundamentalmente diferentes das situações anteriores.

2.3.5 Replicação

As soluções baseadas em *software* de tolerância a faltas em sistemas distribuídos consistem na *replicação* do mesmo serviço em vários processos, para

assegurar que o serviço se mantém disponível apesar de eventuais faltas. A complexidade da concretização deste tipo de soluções provém da necessidade de manter as réplicas coerentes mesmo na presença de faltas.

Um critério de coerência para um servidor replicado é que a história da sua computação seja *linearizável* [HW90], o que informalmente significa, que as histórias das várias réplicas possam ser interpretadas como uma única história que os clientes do serviço vêm como uma computação correcta de um servidor não replicado.

As estratégias de replicação activa ou de máquinas de estados determinísticas [Sch93b] ou então passiva ou primário-secundários [BMST93] são duas técnicas que asseguram a coerência forte ao cumprirem as condições suficientes [GS96b]:

Atomicidade Qualquer invocação processada por uma das réplicas é também processada por todas as réplicas correctas.

Ordem Qualquer par de invocações processadas por quaisquer duas réplicas distintas, é processado por ambas segundo a mesma ordem.

Para assegurar a ordem e a atomicidade é frequentemente utilizado o paradigma da difusão fiável em grupos de processos [Bir93]. Outros paradigmas são transacções atómicas [PSWL95] e modelos de coerência em memória distribuída partilhada [RM93], que podem ser relacionados com difusão fiável [Fri95, GS95].

2.4 Comunicação em grupos de processos

2.4.1 Difusão fiável

Informalmente, difusão fiável significa que uma mensagem ou chega a todos os destinatários ou então não chega a nenhum. Embora não seja óbvio, um sistema onde seja possível a falha de um processo por paragem não oferece quaisquer garantias neste campo, mesmo considerando canais de comunicação fiáveis. A difusão neste modelo passa obrigatoriamente pelo envio sucessivo da mensagem a cada um dos destinatários, cenário em que a falha do processo emissor leva a que eventualmente alguns dos destinatários não recebam a mensagem.

A correcção de um protocolo de difusão fiável resume-se pois ao respeito pelas propriedades [HT94]:

Validade Se um processo correcto difunde uma mensagem, essa mensagem é-lhe entregue num tempo não determinado mas finito.

Unanimidade Se uma mensagem é entregue a um processo correcto, essa mensagem é entregue a todos os processos correctos num tempo não determinado mas finito.

Integridade Qualquer mensagem é entregue a qualquer processo no máximo uma vez e apenas se foi difundida previamente.

2.4.2 Ordenação

Um outro aspecto relevante para a coerencia de um grupo de réplicas é a ordenação das mensagens recebidas. Dependendo dos problemas e dos algoritmos considerados, podem ser tolerados diferentes graus de liberdade na ordenação das mensagens num processo relativamente à sua difusão e à sua entrega noutros processos.

Normalmente, as propriedades de um canal em termos de ordenação são classificados, da mais fraca para a mais forte [HT94]:

Nenhuma ordem A ordem de entrega é completamente independente da ordem de difusão.

Ordem FIFO Se um processo correcto difunde uma mensagem m antes de uma outra m', então a nenhum processo correcto é entregue m' antes de lhe ter sido entregue m.

Ordem causal Se a difusão de uma mensagem m precede causalmente[2] a difusão de uma mensagem m', então a nenhum processo correcto é entregue m' antes de lhe ter sido entregue m.

Estas classes formam uma hierarquia, no sentido em que invariantes de ordem que têm que ser respeitados numa classe, também são na seguinte. Isto significa que um algoritmo que dependa de uma destas ordenações pode funcionar sobre um canal mais forte e um canal que satisfaça uma delas pode ser usado por algoritmos que dependam de qualquer outro mais fraco.

Nos casos considerados na secção anterior, apenas se abordaram ordens parciais, pelo que era sempre possível a reordenação de algumas mensagens arbitrariamente pelo algoritmo de entrega. Esta reordenação pode ser feita de modo distinto pelos diferentes processos. Porém, a correcção de certas aplicações depende do acordo de todos os processos do sistema quanto à ordem de entrega de mensagens, ou seja, de uma ordem total [HT94]:

Ordem total Se quaisquer dois processos correctos p e q entregam duas mensagens m e m', então p entrega m antes de m' se e só se q entrega m antes de m'.

No modelo considerado, a garantia de uma ordenação total passa pelo estabelecimento de um consenso na entrega de cada mensagem, pelo que a

[2]Um evento precede causalmente outro se a ocorrência do segundo é de algum modo influenciada pelo primeiro. Na prática, normalmente considera-se que o envio de uma mensagem por um processo é causalmente dependente de todas as mensagens recebidas e enviadas previamente pelo mesmo processo.

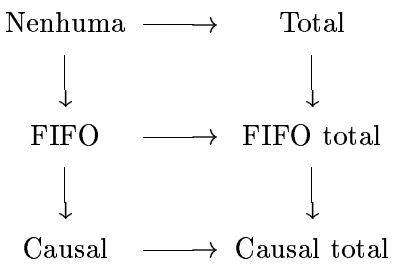


Figura 2.1: Relações entre as classes de ordenação.

sua concretização está limitada às condições onde a resolução do consenso é possível.

A ordenação total quando combinada com as três restantes categorias de ordenação, permite ainda classificar as garantias oferecidas por um canal em seis classes diferentes (Figura 2.1), que são normalmente as usadas para classificar as necessidades de um algoritmo e as disponibilizadas por concretizações de protocolos existentes.

2.4.3 Gestão de grupos

Um outro aspecto relevante para garantir a coerência de um grupo de réplicas é o conhecimento que cada uma delas tem da composição do grupo. Este conhecimento, além de ser necessário por motivos de enumeração e endereçamento dos seus membros é fundamental sobretudo para assegurar simultaneamente a vivacidade e segurança do sistema [BG93].

A ocorrência de falhas e a sua reparação são naturalmente razões para que os elementos do grupo se alterem. A incapacidade de reconhecer que uma réplica falhou pode levar o resto do grupo a bloquear, por exemplo, por falta de espaço de armazenamento para assegurar a retransmissão de mensagens para a réplica faltosa. Por outro lado, a possibilidade de detectar como faltosas réplicas pode levar ao mesmo resultado por incapacidade de reunir o quórum necessário para assegurar a correcção do sistema.

A necessidade de garantias quanto ao acordo sobre a composição do grupo surge ainda porque a incoerência a este nível pode fazer com que processos que de outro modo seriam correctos falhem, por exemplo, pela incapacidade de reconhecer o líder num sistema de replicação passiva.

Um mecanismo de gestão do grupo fica encarregado de inserir no fluxo normal de mensagens entregues, informação de gestão do grupo, ou seja, mensagens contendo a vista actual do grupo. O modo como isto é feito depende de um conjunto de pressupostos [HS95]:

- compromisso entre correcção e vivacidade;

- acordo;
- ordem das vistas entre si e com mensagens normais;
- informação suplementar à composição da vista;
- arranque do sistema e recuperação de falhas;
- suporte de partições.

O modelo mais utilizado é o que oferece *sincronismo virtual* [BJ87], que garante ordem total das vistas entre si e acordo total quanto aos sucessores e predecessores de cada mudança de vista.

2.5 Acordo em sistemas distribuídos

2.5.1 Consenso

Subjacente tanto à ordenação de mensagens e gestão de grupo, ao compromisso atómico em transacções e à coerência forte em memória partilhada, está a necessidade de acordo, pelo que a possibilidade de acordo é uma questão fundamental em sistemas distribuídos.

Os diversos problemas que implicam acordo podem ser reduzidos a uma formulação abstracta de acordo em sistemas distribuídos conhecido como o *problema do consenso*. Este problema é definido sobre um conjunto de processos, em que cada um deles propõe inicialmente um valor. Um algoritmo tem então que decidir um valor de modo que seja verdade que:

Unanimidade A decisão de quaisquer dois processos não pode ser diferente.

Validade Se um processo correcto decide um valor, então esse valor foi proposto por algum processo.

Terminação Todo o processo correcto decide exactamente uma vez.

2.5.2 Condições de resolução

A impossibilidade de resolução do problema do consenso por um algoritmo determinístico num sistema assíncrono quando pelo menos um processo pode falhar por paragem foi demonstrada por Fischer, Lynch e Paterson [FLP85], mesmo com canais fiáveis. É no entanto possível resolver o problema assumindo algum sincronismo no sistema [DDS87], ou então usando um algoritmo não determinístico [BO93], que assegura a sua resolução com probabilidade igual a 1.

Tomando em consideração que a dificuldade na resolução do consenso em sistemas assíncronos reside na impossibilidade de detectar falhas de

processos, Chandra e Toueg [CT94] apresentaram o modelo de um sistema assíncrono aumentado com *detectores de falhas imperfeitos*, que apesar de poderem cometer um número infinito de erros tornam solúvel o consenso.

O algoritmo apresentado que resolve o problema do consenso assume um detector da classe $\diamond\mathcal{S}$ definido pelas propriedades:

Totalidade forte Todo o processo que falhe é inevitavelmente suspeitado por todos os processos correctos.

Precisão inevitavelmente fraca Existe um tempo depois do qual algum processo correcto não é suspeitado por nenhum processo correcto.

Foi também demonstrado em [CHT94] que o detector de falhas mais fraco que torna possível a resolução do problema do consenso num sistema assíncrono onde pode falhar pelo menos um processo[3] é o detector da classe $\diamond\mathcal{W}$. Isto significa que, para ser possível efectuar uma decisão distribuída é necessário que as capacidades de detecção de falhas disponíveis no sistema cumpram pelo menos a totalidade fraca[4] e a precisão inevitavelmente fraca que definem a classe $\diamond\mathcal{W}$.

O algoritmo de Chandra e Toueg assume no entanto a disponibilidade de canais de comunicação fiáveis, o que em sistemas reais é ainda um pressuposto demasiado forte, uma vez que a sua simulação a partir de canais com perda justa depende de quantidade de armazenamento ilimitada [BCBT96]. Foi no entanto demonstrado que é possível resolver o consenso em condições semelhantes, mas que se assume apenas a existência de *canais insistentes*, que podem perder um número ilimitado de mensagens e necessitam apenas de memória limitada proporcional ao número de processos [OGS96].

2.6 Sumário

Neste capítulo apresenta-se o conceito de confiabilidade de um sistema, caracterizando este conceito em termos dos critérios de fiabilidade, segurança, disponibilidade e reparabilidade. Argumenta-se então que as questões fundamentais para atingir estes critérios são a caracterização dos cenários de falha e a introdução de redundância.

Aplicando esta estratégia a sistemas de computação distribuídos, descreve-se sucintamente um modelo para o sistema e para as falhas, apresentando grupos de servidores replicados como sendo o modo de introduzir redundância.

A replicação levanta no entanto o problema da manutenção da coerência das diversas réplicas para garantir a equivalência a uma cópia. Para o efeito,

[3] Embora se assuma que há sempre uma maioria de processos correctos.

[4] Totalidade fraca, significa que todo o processo que falha é inevitavelmente suspeitado pelo menos por um processo correcto.

apresenta-se a difusão fiável e ordenada, em conjunto com a detecção coerente de falhas. Subjacente a esta abordagem está a necessidade de acordo, pelo que são descritas as condições necessárias para a resolução do problema abstracto do consenso.

Capítulo 3

Sistemas distribuídos em grande escala

3.1 Motivação

Uma rede de comunicação suportando um sistema distribuído em grande escala é uma colecção de redes locais interligadas, proporcionando a troca de informação entre os nós das várias redes. Em relação a uma rede local isolada, pode caracterizar-se por [BS94, IP94, PS94]:

- dispersão geográfica;
- um elevado número de elementos;
- diversidade.

Acompanhando o aumento exponencial do número de utilizadores de redes em grande escala, tem sido também crescente o número de aplicações nelas disponíveis. Isto acontece não só pela migração de aplicações a partir de redes locais, como também pelo aparecimento de novas aplicações que passam a ser possíveis ou necessárias no contexto de redes globais. Alguns requisitos destas aplicações são: [PS94, VVR94]:

- partilha de informação coerente, de acordo com o objectivo fundamental das redes em grande escala;
- prontidão semelhante à das aplicações locais, considerando tanto a disponibilidade como o desempenho;
- ferramentas de programação adequadas para a migração de aplicações de redes locais e obtenção de sistemas confiáveis;

- extensibilidade e capacidade de evolução, dado o rápido crescimento e inovação em redes em grande escala;
- segurança tanto da informação transmitida como também na informação armazenada.

Para satisfazer estes requisitos torna-se interessante considerar a utilização de protocolos de comunicação em grupo como ferramenta de programação adequada para desenvolver serviços replicados para tolerância a faltas e elevado desempenho.

Porém, há necessidade de se avaliar em que medida as diferenças inerentes à mudança de escala da infra-estrutura de comunicação afectam os sistemas de comunicação em grupo desenvolvidos para redes locais.

O projecto e concretização de sistemas distribuídos confiáveis neste ambiente implica então avaliar da medida em que estes aspectos alteram o modelo assumido para um sistema distribuído, quer relativamente aos pressupostos de correcção como também aos de desempenho.

3.2 Consequências da grande escala

3.2.1 Escala geográfica

A dispersão de um sistema distribuído por uma grande área tem consequências importantes relacionadas sobretudo com os mecanismos usados para interligar diversas redes locais, bem como as respectivas falhas, que invalidam alguns pressupostos assumidos em redes locais [IP94].

A utilização de redes públicas de comutação de pacotes como mecanismos de interligação de redes locais, como são partilhadas por diversos utilizadores tornam difícil a reserva de largura de banda e a garantia de tempos de resposta. Isto faz com que seja também difícil assumir com segurança qualquer tipo de sincronismo, devido não só à imprevisibilidade da carga da rede e dos nós como também à possibilidade de *partição* do sistema em vários sub-sistemas isolados devido a falhas das interligações [BDM95].

Falhas parciais dos mecanismos de interligação ao nível dos mecanismos de encaminhamento de pacotes utilizados, podem também dar origem a que não seja perfeita a simetria e transitividade dos canais de comunicação, levando à existência de *partições parciais* em que o sistema se divide em vários sub-sistemas que mantêm apenas algumas possibilidades de comunicação [IP94, BBD96, Mal95].

Passa também a ser significativa a perda e duplicação de pacotes, que em redes locais é negligenciável em termos de desempenho [IP94]. Além disso, como cada vez que uma mensagem é difundida, a possibilidade de ter que ser retransmitida obriga ao seu armazenamento temporário. Por outro

lado, a recepção de mensagens fora de ordem em consequência do assincronismo do sistema, obriga a que a sua entrega seja atrasada, armazenando-as também temporariamente. Em ambos os casos o espaço de armazenamento necessário cresce e torna-se de difícil previsão.

Em ambos os casos como não pode ser previsto o tempo durante o qual é preciso armazenar as mensagens, e em consequência qual o espaço necessário, tem que se admitir a possibilidade do sistema bloquear se o espaço previsto se revelar insuficiente, o que é agravado num sistema em grande escala onde as perdas e reordenações de mensagens são significativas.

O assincronismo torna também impossível a detecção fiável de falhas de processos, normalmente conseguida através de mensagens de monitorização e de temporizadores.

Outros sistemas que apresentam características semelhantes às consequências da escala geográfica são sistemas de computação nómada, devido à existência de modos funcionamento desligado ou parcialmente ligado por redes de comunicação de qualidades diversas e variáveis.

3.2.2 Escala numérica

A segunda característica de sistemas distribuídos em grande escala é o elevado número de elementos participantes, que causa problemas com algoritmos cuja complexidade em termos de tempo de execução ou em termos de recursos consumidos é proporcional ao número de intervenientes [BS94].

Um exemplo é a ocupação de espaço por vectores e matrizes de números de sequência que torna impraticáveis protocolos de comunicação que os usem quando estão envolvidos algumas centenas de processos [SM94]. Outro exemplo é a dificuldade na entrega de vistas de grupos com grande número de elementos e o seu posterior processamento por parte de cada um dos processos.

Algoritmos de acordo distribuído são também um problema. Os que necessitam de uma troca de mensagens de todos para todos, num sistema com um elevado número de processos requerem uma largura de banda considerável, que normalmente não está disponível, tem um custo demasiado elevado ou demora demasiado para satisfazer os requisitos de prontidão por parte dos utilizadores. Os protocolos baseados na passagem de testemunho, introduzem também uma latência demasiado elevada por ser proporcional ao número de processos [RFV96].

Características semelhantes podem ser encontradas em sistemas paralelos em grande escala, onde o número de processadores e a necessidade de explorar todo o desempenho teoricamente possível são as preocupações fundamentais.

3.2.3 Diversidade

Um sistema em grande escala é ainda caracterizado por uma grande diversidade quer nos computadores como na rede [PS94]:

- capacidade dos computadores, em termos de potência de cálculo, memória principal e memória secundária;
- processador e sistema operativo;
- autoridade e administração dos diferentes componentes;
- largura de banda e frequência de faltas nas interligações.

Esta diversidade dificulta a utilização de soluções uniformes bem como a sua actualização e evolução. A diversidade a nível da autoridade sobre os diferentes componentes tem ainda uma importância fundamental nos aspectos relacionados com a segurança.

3.3 Comunicação em sistemas em grande escala

As propostas de sistemas de comunicação fiável apresentadas para sistemas em grande escala baseiam-se no facto de, proporcionalmente, os recursos em redes em larga escala serem mais escassos e os pressupostos a fazer quanto ao sistema mais fracos. Em resposta a estes problemas encontram-se duas estratégias:

- uma melhor adequação das garantias oferecidas às aplicações, no sentido de estas não serem penalizadas no desempenho por facilidades de que não fazem uso, procurando minimizar o impacto de uma infra-estrutura que oferece uma menor disponibilidade de recursos;
- exploração cuidada de todas as hipóteses de paralelismo entre diferentes tarefas, sobretudo para enfrentar os problemas decorrentes da escala numérica em conjunto com a complexidade dos algoritmos.

O mesmo tipo de preocupações de desempenho encontram-se na computação paralela, pela necessidade de empregar números cada vez maiores de processadores. Por outro lado, preocupações respeitantes à disponibilidade encontram-se também na computação nómada, onde a possibilidade de operação desligada da rede tem efeitos semelhantes às partições em redes em larga escala.

3.3.1 Relaxamento da fiabilidade

Um primeiro conjunto de propostas pretende diminuir os recursos necessários ao armazenamento e retransmissão de mensagens.

As *mensagens transparentes à causalidade* [RV94] são normalmente entregues causalmente ordenadas. No entanto, como nenhuma mensagem pode ver a sua entrega atrasada por uma mensagem transparente, estas podem nunca ser entregues se entretanto já foi entregue uma sucessora.

Os canais *k-insistentes* apenas garantem a entrega das k últimas mensagens difundidas por cada processo [GOS96], sendo demonstrado que é possível resolver o problema do consenso relaxando a necessidade de canais fiáveis para canais 1-insistentes.

De facto, as mensagens transparentes à causalidade constituem um boa aproximação aos canais 1-insistentes, no caso em que todas as mensagens emitidas por qualquer processo são classificadas como transparentes. Não são no entanto equivalentes, dada a relação entre mensagens de diferentes emissores.

3.3.2 Relaxamento da ordenação

Outras propostas tentam reduzir os atrasos de mensagens devido a dependências de ordem, ao mesmo tempo diminuem a complexidade dos algoritmos de ordenação.

A mais simples resume-se a etiquetar as mensagens enviadas com o tipo de ordenação a que deverão estar sujeitas, o que já é feito em protocolos para redes locais. Por exemplo, no XAMP [RV92] é possível especificar se a mensagem é FIFO, causal ou total, garantindo que a entrega dessa mensagem respeita todas as restrições dessa ordenação em relação a quaisquer outras mensagens.

Outra abordagem semelhante sugere uma classificação de cada uma das mensagens como *causal* ou *ordinária* [MR94]. Qualquer par de mensagens em que uma delas é causal, é entregue devidamente ordenado. Pares em que ambas são ordinárias podem ser entregues por qualquer ordem. É proposta também a etiquetagem de mensagens como *serializadas*, que se comportam como causais mas com a restrição da entrega ser totalmente ordenada.

Uma proposta que permite relaxar o ordenação de mensagens para o mínimo necessário é a definição dos passados causais[1] directamente pela aplicação, apresentada no contexto do sistema de replicação *preguiçosa* [LLS91]. Ao contrário de outras soluções onde as mensagens são ordenadas pela causalidade potencial, deste modo apenas se atrasam mensagens devido a outras que efectivamente pertençam à sua história causal, reduzindo a possibilidade de bloqueio.

[1] Ver Capítulo 5 para uma definição formal de "passado causal". Informalmente define-se como passado causal de um evento o conjunto de eventos que, directa ou indirectamente, lhe dão origem.

3.3.3 Coerência fraca em memória partilhada

No contexto de sistemas de memória partilhada distribuída, o relaxamento das garantias oferecidas pelo sistema de memória pode ser feito segundo modelos de coerência de memória híbridos, que generalizam diferentes tipos de modelos de coerência fraca [AF92].

Um modelo de coerência hibrido define-se pela existência de dois tipos de operações sobre a memória, denominadas fortes e fracas, sendo verdade que:

- todas as operações fortes são observadas por uma ordem equivalente a uma ordem sequêncial;
- se duas operações são executadas pelo mesmo processo ou uma delas é forte, então são observadas em qualquer processo pela ordem de execução.

Se todas as operações são fortes então obtém-se um modelo de coerência sequêncial. Se todas as operações são fracas então obtém-se o modelo de coerência mais fraco possível. Mais interessantes são os modelos intermédios, em que algumas operações são fortes e outras são fracas. Um exemplo é a coerência à saída [KCZ92] em que as operações sobre variáveis de exclusão são fortes e as restantes operações são fracas.

Uma vez que o comportamento de um sistema de memória partilhada é definido em termos dos resultados observados e não em termos das operações efectuadas, um modelo de coerência hibrida resulta quer num relaxamento da ordem pela qual as actualizações podem ser feitas, bem como num relaxamento de quais as actualizações são de facto efectuadas. Por exemplo, num sistema de coerência à saída, todas as actualizações menos a última antes de uma operação de sincronização podem não ser efectuadas.

3.3.4 Diferentes tipos de processos

É ainda possível efectuar um relaxamento das garantias de qualidade de serviço com base numa classificação dos processos nos sistemas Relacs [BDGB95] e Phoenix [Mal96]. Neste sistema os processos são classificados como *passivos*, *clientes* e *membros* [BS94]. Os primeiros limitam-se a receber mensagens. Os clientes podem também enviar mensagens para um ou mais grupos, embora não façam parte das vistas entregues nesses grupos, papel reservado para os membros de cada grupo.

Este mecanismo permite ainda resolver os problemas relacionados directamente com a escala, pois ao reduzir o número de processos a serem tratados como membros de pleno direito, evita o crescimento exponencial da complexidade dos algoritmos necessários, por exemplo, para chegar a acordo sobre uma nova vista do grupo.

3.3.5 Múltiplos grupos

Uma técnica usada para contornar o mesmo problema é a utilização de vários grupos, com membros comuns, sendo as mensagens difundidas nos diferentes grupos para obter diferentes tipos de garantias.

Os problemas desta abordagem surgem quando é necessário oferecer garantias simultâneas quanto à ordenação relativamente a mais do que um grupo. Neste caso, a maior parte dos sistemas não oferece qualquer suporte deixando essa tarefa entregue à aplicação, embora existam várias propostas no sentido de suportar garantias entre diferentes grupos à custa de protocolos mais complexos para ordenação total [SG97] e causal [MR93]. Um sistema que oferece este tipo garantias quanto à ordenação total de mensagens em grupos sobrepostos é o NEWTOP [EMS95].

3.3.6 Grupos particionáveis

As falhas nas interligações podem dar origem a sistemas em que apesar de existir uma maioria de processos correctos, estes não podem comunicar entre si. Nestes casos, um sistema de sincronismo virtual que apenas instale vistas maioritárias, ou seja, contendo uma maioria do total dos processos do sistema e em que processos incluídos em vistas minoritárias sejam terminados, pode levar a um bloqueio definitivo do sistema no caso de não ser possível reunir uma maioria.

O sistema PHOENIX, sendo baseado num protocolo de consenso, contorna o problema nunca instalando vistas minoritárias, obrigando a um bloqueio temporário e retomando a operação logo que a atingibilidade de uma maioria esteja assegurada.

Os mecanismo de gestão de grupos presente nos sistemas TRANSIS, TOTEM e HORUS [DMS96, MAMSA94] e no sistema RELACS [BDM95] permitem a instalação de vistas minoritárias concorrentes, assegurando no entanto que:

- todos os processos que instalam duas vistas sucessivas entregam as mesmas mensagens entre elas;
- todos os processos que instalam uma mesma vista têm uma visão coerente do sistema, em termos das vistas instaladas no passado.

Esta solução, com a consequente possibilidade de operação independente implica no entanto a necessidade de reconciliar o estado das diversas partições quando estas se fundem. Para o efeito o RELACS fornece à aplicação informação adicional sobre a história das partições a fundir, identificando a partição de que são originários e deixando à aplicação a decisão final de quando efectuar as fusões [BBD96].

3.3.7 Hierarquia

Um modo de esconder a escala numérica consiste na divisão dos processos em vários grupos hierárquicos. Uma proposta neste sentido é hierarquização em dois níveis, fazendo a distinção entre processos e nós da rede, economizando tempo e espaço na detecção de falhas e instalação de vistas [VRV92].

A hierarquização em diversos níveis é também usada para a difusão [DM96], difusão fiável [GvRVB97], entrega causalmente ordenada [BFvR96] e entrega totalmente ordenada de grupos aninhados [AMA+95], permitindo diminuir a informação de controlo necessária e paralelizar as operações nos diversos sub-grupos.

3.3.8 Topologia da rede

Como uma rede em grande escala não é uniforme, pode ser vantajoso para um sistema reconhecer as diferenças entre diferentes segmentos da rede e entre as ligações dos vários pares de processos.

Uma proposta neste sentido procura minimizar a informação sobre causalidade armazenada quer nas mensagens quer nos processos [RV95], o que é importante uma vez que com um número elevado de processos o espaço necessário para armazenar esta informação é significativo. Isto é conseguido separando a rede em diversos componentes e filtrando a informação contida nas mensagens quando atravessam as fronteiras, de modo a restringir a sua disseminação apenas aos componentes onde é necessária.

Para efeitos de desempenho é também conveniente reconhecer que um sistema em grande escala é caracterizado como um conjunto de redes locais onde a difusão de mensagens é possível, interligadas por ligações ponto-a-ponto onde a difusão é simulada por envios sucessivos [MMSA+96, DM96].

3.4 Sumário

Nesta secção introduziu-se a necessidade de alargar a disponibilidade de ferramentas de programação de sistemas distribuídos confiáveis a redes em grande escala. Os problemas resultantes deste alargamento relacionam-se com a escala geográfica, a escala numérica e a diversidade a diversos níveis características de redes em grande escala.

Como resposta, apresentam-se diversas propostas, como o relaxamento das garantias de fiabilidade, de ordenação e de gestão de grupos, em conjunto com a adequação à estrutura das redes em grande escala. Destas propostas salientam-se as ideias comuns de rever a utilidade das garantias dadas às aplicações e de procurar explorar todas as oportunidades de paralelismo existentes no sistema.

A revisão das garantias que é necessário oferecer a uma aplicação tem como objectivo proporcionar-lhe apenas aquelas que são estritamente ne-

cessárias, de modo a não a penalizar por garantias que não usa. A exploração das oportunidades de paralelismo surge como complemento, possibilitando a utilização de sistemas distribuídos de escalas cada vez maiores.

Capítulo 4

Concretização de protocolos

4.1 Motivação

O projecto e concretização de protocolos de comunicação em geral e de protocolos de comunicação fiável em particular é uma tarefa complexa devido a vários factores:

- têm um papel central em sistemas distribuídos, não só em termos de funcionalidade como também de desempenho;
- estão sujeitos a falhas da rede, em termos de atrasos, perdas e duplicações de pacotes e ainda partições;
- estão sujeitos a falhas dos processos intervenientes;
- têm que ser adaptáveis de forma independente às modificações da rede de comunicação e às necessidades das aplicações, de modo a evoluir com ambos.

De modo a resolver estes problemas, têm sido usadas diversas estratégias para a modularização de protocolos de comunicação, quer por simples estratificação, quer por programação orientada por objectos, até à migração de código.

Além de poderem ser utilizadas para desenvolver protocolos independentes, estas estratégias são normalmente concretizadas sob a forma de infra-estruturas de *software* que oferecem funcionalidade comum a diversos protocolos, tais como:

- mecanismos para definição e composição de módulos de protocolos;
- manipulação de recursos do sistema operativo;
- controladores de dispositivos;

- interface com as aplicações.

Na escolha de uma infra-estrutura além dos diversos aspectos técnicos que é importante considerar, tais como:

- adequação aos protocolos a desenvolver;
- disponibilidade da infra-estrutura em diversas plataformas;
- possibilidade de inter-operação entre diferentes módulos de um mesmo protocolo e entre protocolos e aplicações, possivelmente usando diferentes linguagens.

é importante ainda considerar disponibilidade de código-fonte da infra-estrutura e condições de redistribuição de protocolos que a utilizem.

4.2 Arquitecturas tradicionais

4.2.1 Pilhas de protocolos

A forma mais comum para estruturar protocolos de comunicação é a sua divisão em camadas empilhadas, normalmente conhecidas como pilhas de protocolos. Deste modo, é possível utilizar apenas parte da pilha, ou camadas alternativas, de acordo com o tipo de serviço que se pretende para cada aplicação. É esta a abordagem sugerida pelo modelo de referência OSI [DZ83]. Em protocolos de grupos, esta estruturação encontra-se por exemplo no PHOENIX [Mal96].

A separação em camadas é efectuada de modo a proporcionar diferentes níveis de abstracção sobre o sistema de comunicação. A norma OSI especifica para protocolos ponto-a-ponto, por exemplo, que a um nível se veja a rede como um mecanismo de passagem de mensagens entre nós directamente ligados, que no nível seguinte se esconda o encaminhamento e se ofereça passagem de mensagens em qualquer ponto da rede e em seguida se ofereça um fluxo continuo e controlado de dados. Em termos de protocolos de difusão, no Phoenix encontram-se camadas para difusão de mensagens não fiável, difusão não fiável com encaminhamento, difusão fiável, sincronismo virtual e difusão ordenada.

No que diz respeito à concretização deste tipo de protocolos, cada camada funciona como um filtro, aceitando uma mensagem vinda da camada imediatamente acima, da qual um caso especial é a aplicação, acrescenta cabeçalhos respeitantes à função que desempenha e envia-a para a camada imediatamente inferior, da qual um caso especial é a rede física. Quando a mensagem é recebida, em cada camada os cabeçalhos respectivos são examinados e removidos da mensagem, sendo esta posteriormente entregue à camada superior.

Além de acrescentar e remover cabeçalhos, cada camada pode ainda fragmentar, reagrupar, reter, duplicar ou mesmo suprimir mensagens. No entanto, para manter a independência entre as diferentes camadas, é importante que os cabeçalhos acrescentados a uma mensagem numa camada de protocolo sejam:

- ignorados pelas camadas inferiores, ou seja, tratados como parte dos dados;
- manipulados apenas pela camada de protocolo em que foram criados;
- retirados antes da mensagem ser entregue ao nível superior durante a recepção.

Deste modo as diferentes camadas ficam dependentes apenas em termos de especificação do serviço prestado pelo nível inferior e não de qualquer concretização particular de quaisquer outras camadas.

Duas questões importantes tanto para o desempenho de uma pilha de protocolos como para a complexidade do seu desenvolvimento são a utilização actividades concorrentes e o número de cópias da mensagem efectuadas. Em termos de actividades concorrentes podem identificar-se duas alternativas:

- uma actividade por cada camada de protocolo;
- uma actividade por cada mensagem.

No primeiro caso evita-se o uso de mecanismos de controlo de concorrência dentro de cada camada, à custa da penalização do desempenho devida à sincronização entre cada duas camadas correspondente a cada mensagem trocada.

As necessidade de acrescentar cabeçalhos como prefixos às mensagens em cada camada, em conjunto com o armazenamento de mensagens como sequências contíguas de dígitos binários, dá origem a que:

- cada camada de protocolo corresponda a uma cópia de toda a mensagem para um novo espaço de armazenamento temporário com a dimensão adequada, o que tem um considerável impacto no desempenho;
- seja utilizada uma estrutura de dados ligada de fragmentos da mensagem, incorrendo num custo extra em complexidade e desempenho para gerir essa estrutura.

É de referir que mesmo a segunda não é completamente satisfatória, dada a necessidade de converter a estrutura ligada numa sequência contigua através de uma cópia quando se atravessa uma fronteira de protecção, por exemplo, entre a aplicação e o núcleo do sistema operativo ou entre o sistema operativo e o dispositivo de rede.

4.2.2 Pilhas de protocolos uniformes

Uma evolução das pilhas de protocolos é feita pela uniformização da interface entre as camadas, de modo a poderem ser empilhadas arbitrariamente camadas de diferentes origens, o que é conseguido pelo X-KERNEL [HP91] e pelo HORUS [vRB95, vR96]. Isto faz com que esta estratégia seja especialmente adequada ao desenvolvimento de infra-estruturas genéricas de desenvolvimento de protocolos.

Para que cada camada seja um componente que se pode combinar, a uniformização tem que ser feita a vários níveis:

- normalização de facilidades disponibilizadas pelo sistema operativo, tais como gestão de actividades, de memória e de eventos tais como temporizadores;
- normalização das operações importadas e exportadas por cada camada de protocolo, incluindo tipicamente operações de envio e recepção de mensagens de outras camadas;
- normalização de procedimentos de instalação de pilhas de protocolos e de inicialização de sessões de comunicação;
- utilização de estruturas de dados comuns para representar objectos partilhados pelas diversas camadas, como por exemplo, as mensagens em trânsito e identificadores de participantes.

Além de encorajar a independência entre camadas de protocolo, com as consequentes vantagens em termos de projecto, manutenção e reutilização, permite a concretização de camadas de protocolo virtuais [OP92]. Camadas de protocolo virtuais não acrescentam cabeçalhos às mensagens, o que faz com que não comuniquem com o seu par no outro extremo do canal de comunicação e portanto que não sejam verdadeiros protocolos de comunicação. Porém, são úteis, por exemplo, para encaminhar mensagens na pilha de protocolos ou para obter estatísticas sobre um fluxo de mensagens, funcionando a qualquer nível de qualquer pilha, o que não seria possível sem a normalização.

O projecto de uma infra-estrutura de protocolos uniformes tem no entanto um problema de maior a resolver, relacionado com as operações que são incluídas na interfaces entre diferentes camadas. A escolha destas operações resulta de um compromisso entre:

- um conjunto mínimo de operações como reflexo de pressupostos também mínimos sobre os tipos de protocolos suportados, mas que torne difícil a coordenação entre camadas de protocolos que necessitem de trocar informação suplementar;

- um conjunto de operações mais extenso e adequado a um tipo específico de protocolos, nomeadamente prevendo operações específicas para esses protocolos trocarem informação entre si, mas tornando mais complexa a concretização de cada camada e a infra-estrutura de aplicação mais restrita.

Um exemplo deste compromisso é o GTS [MBM95], que inclui apenas um conjunto mínimo de operações entre camadas, obrigando a violar o princípio de independência entre camadas ao ter operações especiais entre algumas camadas.

Um exemplo contrário é a inclusão no x-Kernel de operações respeitantes a protocolos com semântica de invocações remotas e operações com semântica de passagem de mensagens, que apesar de raramente fazerem sentido no mesmo protocolo, terem que ser todas concretizadas. Outro exemplo é ainda a inclusão no Horus de operações respeitantes à manutenção de sincronismo virtual num grupo, o que torna inadequada a sua utilização para protocolos ponto-a-ponto bem como de protocolos de difusão que não incluem sincronismo virtual.

4.2.3 Micro-protocolos

Como as camadas de protocolo tendem a ser ainda componentes bastante complexos, os sistemas Consul [Hil96] e xAMp [Fon94] propõem a construção de camadas de protocolo por agregação de micro-protocolos baseados em eventos, no sentido de aumentar a possibilidade de modularização, reutilização e reconfiguração de cada camada de protocolo.

Esta arquitectura baseia-se na compreensão de protocolos como máquinas de estados, transitando de estado em resposta a eventos, como a chegada de uma mensagem ou de sinais de um temporizador. Podem então identificar-se quais os estados e as transições relacionadas com cada aspecto de um protocolo, isolando-os dos restantes e reunindo-os num módulo, a que se chama um micro-protocolo. A limitação desta arquitectura reside precisamente na dificuldade em separar os micro-protocolos de modo a poderem ser compostos sem interferirem de formas inesperadas.

Em termos de concretização, isto significa que cada protocolo é constituído por um conjunto de micro-protocolos que partilham uma estrutura de dados. Cada um dos micro-protocolos é um conjunto de procedimentos que são executados em resposta a eventos. Os eventos podem ser pré-definidos pelo sistema, por exemplo, a chegada de uma mensagem ao protocolo, ou especialmente adaptados para a comunicação entre micro-protocolos relacionados. Estes procedimentos podem manipular os dados partilhados do protocolo, dados locais ao micro-protocolo, registar e remover associações de procedimentos a eventos e mesmo despoletar outros eventos.

4.2.4 Linguagens especializadas

Uma outra estratégia consiste no desenvolvimento de linguagens especiais para descrever protocolos de comunicação, recorrendo a compiladores que conseguem optimizar as operações e estruturas mais comuns ou mais críticas no desenvolvimento de protocolos.

A linguagem MORPHEUS [AP93] disponibiliza como construções da linguagem as entidades do sistema em camadas X-KERNEL. Deste modo é possível não só herdar parte da definição de camadas de alguns protótipos disponíveis, chamados *formas*, bem como tornar negligenciavel a sobrecarga introduzida pela estratificação em camadas.

Um outro exemplo é o ADAPTIVE [SBS93] que combina módulos de protocolos, segundo descrições da qualidade de serviço pretendida, gerando protocolos especialmente optimizados às necessidades de cada aplicação. Deste modo é possível reter no código gerado, que determina o desempenho final, apenas as facilidades dos diversos módulos que são utilizados, sem que isso seja conseguido a partir de uma decomposição em módulos demasiado pequenos e numerosos para serem eficientes.

A introdução de novas linguagens tem no entanto alguns inconvenientes:

- representa uma curva de aprendizagem mais acentuada para um programador;
- introduz problemas no suporte a múltiplas plataformas, restrito pela disponibilidade do compilador;
- dificulta a integração com outros módulos escritos em outras linguagens, nomeadamente, com as aplicações;
- a flexibilidade introduzida na configuração estática de protocolos é perdida em parte pela impossibilidade de reconfigurar os protocolos dinamicamente.

4.3 Arquitecturas orientadas por objectos

4.3.1 Classes de protocolos

Como em outras áreas, métodos de programação orientada por objectos servem para diminuir a complexidade do desenvolvimento de protocolos de comunicação. Deste modo, cada um dos intervenientes de uma sessão de comunicação é representado como um objecto de uma classe correspondente ao protocolo em questão [GFG96a].

Como consequência, novos protocolos podem ser definidos como especializações ou extensões de classes existentes, desenvolvendo hierarquias de protocolos, dos quais um programador de aplicações selecciona o mais conveniente, quer para utilização imediata como para especialização.

Esta estratégia resulta portanto num modelo de programação especialmente bem integrado com a programação de aplicações orientadas por objectos, onde os serviços de comunicação oferecidos pelo sistema estão perfeitamente integrados com os restantes serviços bem como com a aplicação em si.

Em contrapartida, esta abordagem torna impossível a reutilização das classes derivadas independentemente das suas classes base, o que é problemático sobretudo quando a complexidade das extensões é superior ao da própria base, como acontece por exemplo com protocolos de comunicação fiável baseados em passagem de mensagens simples [GFG97].

4.3.2 Composição de protocolos e padrões

Como alternativa à herança como mecanismo base para modificar e construir protocolos complexos, é possível construir classes que isolam uma funcionalidade específica encontrada em diversas situações na construção de um protocolo de comunicação [HJE95, SBS93]. Esses objectos elementares podem então ser compostos em protocolos mais complexos.

Relativamente à modularização de protocolos em camadas e micro-protocolos uma abordagem orientada por objectos é mais flexível porque não requer a uniformidade das interfaces entre os diferentes objectos. Deste modo podem ser identificados diferentes papeis a ser desempenhados dentro de um protocolo, como por exemplo, multiplexagem de fluxos de dados, inicialização de sessões, gestão de eventos e actividades, construindo diversas opções para cada um deles.

Um elemento chave para o sucesso desta estratégia é a utilização de padrões arquitecturais conhecidos [GHJV95], como guias para identificar os componentes elementares relevantes bem como a forma como podem ser combinados [Sch93a, GFG96b, HJE95].

4.3.3 Concretização aberta e reflexão

As técnicas de reflexão e concretização aberta [KP96, Kic92], propostas inicialmente no contexto de linguagens de programação são também úteis na programação de protocolos de comunicação orientados por objectos.

A concretização aberta de um módulo de *software* significa que um conjunto seleccionado de características da concretização é exposto ao cliente do módulo, por oposição a uma concretização ocultada em que nenhum detalhe é visível. Isto é feito através de uma interface separada da interface funcional, a meta-interface, através da qual é possível modificar o comportamento do módulo tanto em termos de funcionalidade como de desempenho.

Num sistema orientado por objectos, a concretização aberta resulta em que as entidades que descrevem o comportamento de um objecto sejam também objectos, que podem ser utilizados pelo programador de aplicações

do mesmo modo que manipula quaisquer outros objectos. A estes objectos que fazem parte da meta-interface dá-se então o nome de meta-objectos.

Como consequência, diz-se que o funcionamento do sistema de objectos está reflectido em si próprio, dando o nome de reflexão a este paradigma de programação.

Na programação de protocolos de comunicação faz sentido tomar um protocolo como uma meta-sessão de comunicação e como tal considerar o objecto que concretiza esse protocolo como um meta-objecto da sessão de comunicação. Existem então várias propostas no sentido de permitir ao programador de aplicações instalar e configurar dinamicamente protocolos de comunicação criando e instalando diferentes meta-objectos [AFPS92, FNP+95, McA95].

4.3.4 Mensagens inteligentes

Uma outra abordagem ao desenvolvimento de protocolos orientados por objectos é considerar as mensagens trocadas como objectos activos serializados, que são reconstituídos quando chegam ao destino, onde são executados e respondem a invocações. A esta abordagem dá-se o nome de *mensagens inteligentes* [AOW96].

Esta estratégia tem a vantagem em relação às anteriores de tratar como entidades independentes os dados armazenados nos protocolos e os dados acrescentados às mensagens. Esta separação obtida com a ocultação das estruturas de dados dos cabeçalhos permite o desenvolvimento, manutenção e evolução em separado, acrescentando uma nova dimensão à modularização de protocolos.

Como consequência, é uma alternativa interessante à estrita normalização das estruturas de dados das mensagens para obter inter-operação entre diferentes concretizações do mesmo protocolo. Esta hipótese é no entanto de utilidade limitada enquanto for necessário instalar em cada um dos nós intervenientes o código correspondente à classe dos objectos utilizados como mensagens.

4.4 Redes activas

4.4.1 Definição

O conceito de *redes activas* é definido por Tennenhouse e Wetherall [TW96] como redes em que é possível:

- os nós da rede actuarem sobre a informação que circula na rede, examinando-a e possivelmente modificando-a, não estando restritos a actuarem sobre os cabeçalhos acrescentados por protocolos de comunicação;

- os utilizadores configurarem as computações efectuadas pela rede sobre a informação em trânsito, introduzindo código executável nos diversos nós da rede.

Esta capacidade está intimamente ligada com a inclusão de código executável móvel nos próprios pacotes, que pode ser instalado e executado em nós remotos.

Um primeira abordagem às redes activas consiste na instalação de protocolos de comunicação quer nos extremos de uma sessão de comunicação bem como nas interligações. Uma proposta neste sentido é o ambiente ComScript [MMT94] que permite instalar no extremo oposto de uma sessão de comunicação protocolos desenvolvidos numa linguagem interpretada. Outra proposta semelhante é o Switchware [SFG+96].

Uma abordagem extrema é o caso em que cada mensagem é apenas um programa que é executado em cada nó em que passa, estando os dados a comunicar embebidos nesse programa. Deste modo, o protocolo é completamente concretizado no código contido nas próprias mensagens, conhecidas como *mensageiros* [MMTH95] ou *cápsulas* [TSS+97].

4.4.2 Discussão

Em relação às outras abordagens, baseadas em diversas estratégias de modularização, a possibilidade de migrar código como parte integrante de protocolos de comunicação que é oferecida pelos diversos tipos de redes activas é vantajoso em duas situações:

- actualização de protocolos de comunicação instalados em grande escala, o que torna esta tecnologia atraente para uma futura geração da família de protocolos IP [WT96];

- possibilidade de adaptação às necessidades de aplicações, mais evidente por o protocolo poder ser modificado durante uma sessão de comunicação, o que é importante para um sistema operativo genérico [TMMH94].

Em contrapartida, a abordagem extrema dos mensageiros e das cápsulas só por si não encoraja a modularização dos protocolos em termos de separação de aspectos distintos de um mesmo protocolo de comunicação como é obtido com estratégias mais convencionais.

Esta abordagem pode ainda revelar-se como problemática em termos de segurança dos nós da rede, enquanto executam código proveniente de outros nós possivelmente hostis.

Em termos de desempenho, a inclusão de código nas mensagens representa uma sobrecarga de utilidade limitada, assumindo que em casos normais um nó terá que processar numerosos pacotes carregando todos o mesmo código.

4.5 Sumário

Sendo a funcionalidade e o desempenho de protocolos de comunicação fundamentais para a funcionalidade e o desempenho do sistema distribuído como um todo, apresenta-se neste capítulo um conjunto de arquitecturas para modularizar o projecto e a concretização de protocolos de comunicação, no sentido de ultrapassar a complexidade inerente ao desenvolvimento de protocolos de comunicação.

A utilização das diversas estratégias para modularizar o projecto e a concretização de protocolos faz com que seja possível desenvolver infra-estruturas genéricas para suportar o desenvolvimento de protocolos. A utilização de uma infra-estrutura proporciona um conjunto de facilidades pré-definidas, que simplificam a concretização de protocolos.

A utilização de uma infra-estrutura é também essencial para assegurar a possibilidade de transportar protocolos para diversas plataformas e as aplicações que os utilizam para vários protocolos.

Capítulo 5

Especificação de protocolos configuráveis

5.1 Objectivos

As dificuldades no desenvolvimento de serviços tolerantes a faltas por replicação podem ser agrupadas segundo duas naturezas distintas:

- avaliar quais são as garantias necessárias ao bom funcionamento e exprimi-las em termos de programas;
- assegurar as garantias julgadas necessárias.

Numa rede local é em parte possível aliviar o esforço na correcta avaliação do sistema, pois é possível ser pessimista e investir na disponibilização de garantias mais fortes do que as necessárias. Por exemplo, a utilização de protocolos de fiabilidade e ordenação total como solução genérica para um largo espectro de problemas, uma vez que são condições suficientes para assegurar a linearizabilidade de qualquer sistema de replicação, proporciona um modelo de programação extremamente simples mas capaz de resolver problemas complexos.

No contexto de sistemas distribuídos em grande escala este compromisso é invalidado, sobretudo pelo impacto no desempenho, pelo que surge a necessidade de configurar os serviços de comunicação fiável de modo a, em cada sistema, ser possível assegurar as condições suficientes e necessárias para a linearizabilidade. Para o efeito, é necessário averiguar quais são os parâmetros para especificação das necessidades mínimas das aplicações em termos de comunicação confiável.

Num sistema de ficheiros replicado, duas operações de escrita numa mesma posição de um mesmo ficheiro, sem que ocorram operações de leitura

entre elas, podem ser substituídas por apenas a última operação de escrita. E mesmo que ocorram leituras entre operações de escrita, pode ainda considerar-se a possibilidade de algumas das réplicas apenas executarem a segunda operação de escrita, desde que esteja assegurado que as leituras são efectuadas das réplicas que efectuam ambas as operações de escrita, por exemplo, por restrições de ordem.

Num exemplo semelhante, duas operações de escrita causalmente independentes e em duas regiões disjuntas de quaisquer ficheiros, podem ser efectuadas por réplicas diferentes segundo ordens também diferentes. No entanto, é necessário que duas operações de escrita relacionadas possam ser reordenadas por clientes diferentes por ordens diferentes apenas se o resultado final da computação distribuída for equivalente.

O mesmo se passa com o contexto em termos de vista de grupo em que uma mensagem é entregue. Considerando um exemplo em que a vista é usada para eleger um coordenador, a entrega de uma mensagem pode ser efectuada indiferentemente antes ou depois de uma mudança de vista no caso em que ambas as vistas levam à eleição inequívoca do mesmo processo como coordenador. Ou mesmo em relação à decisão de efectuar uma mudança de vista, a utilização de suspeitas como critério para iniciar a mudança, pode não ser o óptimo. Por exemplo, pode ser mais sensato, dependendo da aplicação, utilizar um critério sobre o espaço utilizado por mensagens não transmitidas ou mesmo um critério que envolva a semântica da aplicação, como só tentar expulsar um processo se é ele que detém um trinco.

De exemplos como estes pode concluir-se que existem situações em que as aplicações permanecem correctas apesar de lhes serem oferecidas garantias mais fracas. No entanto, fica também claro, que as garantias que podem ser relaxadas dependem da semântica de cada aplicação, não sendo possível apresentar soluções genéricas.

Como consequência, a especificação de protocolos em termos de comportamento de uma sessão de comunicação será sempre inadequada uma vez que para ser genérica tem que ser independente da semântica de cada mensagem.

Para resolver este problema, neste capítulo procura especificar-se o comportamento de um protocolo de difusão em termos do seu comportamento em separado para cada mensagem e processo e não em termos do comportamento de uma sessão como um todo.

5.2 Modelo do sistema

5.2.1 Geral

Para um sistema distribuído em grande escala, considera-se um modelo assíncrono constituído por um conjunto de n processos sequenciais, $P =$

$\{p_1, p_2, \ldots, p_n\}$, comunicando por passagem de mensagens numa rede completamente ligada. O assincronismo significa que não pode ser estabelecido qualquer limite superior quer para o tempo entre o envio e a recepção de cada mensagem, quer para a diferença entre as velocidades dos processos. Assume-se ainda a disponibilidade de um detector de falhas da classe $\diamond\mathcal{S}$.

Este modelo é baseado no modelo proposto por Chandra, Hadzilacos e Toueg [CT94], com as seguintes alterações:

- admite-se a perda justa e a duplicação finita de mensagens pelos canais de comunicação;
- assume-se a disponibilidade de um serviço de consenso.

É possível assumir a disponibilidade de um serviço de consenso tendo em conta que a transformação de um canal com duplicação e perda justa num canal 1-insistente, em conjunto com um detector de falhas imperfeito da classe $\diamond\mathcal{S}$, é suficiente para resolver o consenso num sistema assíncrono em que uma maioria de processos não falha [GOS96].

5.2.2 Algoritmos

A computação efectua-se em *passos* de um *algoritmo*. Um algoritmo a é um conjunto de n máquinas de estados determinísticas, uma para cada processo, cada uma com possivelmente um número infinito de estados. a_i é a máquina de estados correspondente ao processo p_i. Em cada passo de a_i, um processo p_i pode:

- mudar apenas o seu estado interno;
- comunicar com outro processo, inserindo ou removendo uma mensagem de um canal;
- iniciar ou concluir a resolução de um consenso com os restantes processos.

Como se assume um sistema assíncrono, não pode ser estabelecido qualquer limite superior para o tempo de execução de cada passo.

5.2.3 Canais

Cada canal unidireccional $c_{i,j}$ de p_i para p_j é modelado como um conjunto de mensagens, modificado pelos processos através da execução dos passos $envia_i(m)$ e $recebe_j(m)$. O passo $envia_i(m)$ executado pelo processo emissor p_i insere a mensagem m no canal e $recebe_j(m)$ executado pelo receptor remove a mensagem do canal. Para o passo $recebe_j(m)$ possa ser executado num processo, é necessário que a mensagem m tenha sido enviada e que

esse processo esteja pronto a recebê-la. De outro modo, ou o processo fica bloqueado ou a mensagem é atrasada, respectivamente.

Como consequência do assincronismo, não pode ser estabelecido qualquer limite superior para o tempo que decorre entre o envio de uma mensagem e a sua recepção.

5.2.4 Histórias

A *história local* de uma execução do processo p_i durante uma computação, é o conjunto infinito dos passos de a_i executados[1], $H_i = \{e_i^1, e_i^2, \dots\}$, totalmente ordenado pela relação $\xrightarrow{i}$ dada pela execução do processo p_i, denominada de *causalidade potencial*.

Uma *história global* de uma computação é um conjunto $H = H_1 \cup \ldots \cup H_n$, ordenado parcialmente pela relação $\rightarrow$, definida como a reunião das ordens das histórias locais com os pares definidos por:

$$\forall m, envia(m) \rightarrow recebe(m)$$

Se para dois passos e e e', não é verdade que $e \rightarrow e'$ nem que $e' \rightarrow e$ então diz-se que esses passos são *concorrentes*, $e \| e'$. Um subconjunto de $C \subseteq H$ tal que:

$$e \in C \wedge e' \rightarrow e \Rightarrow e' \in C$$

diz-se um *corte coerente* de H. Um corte coerente C_e de H que é constituído pelos passos e' tais que $e' \rightarrow e$ dá-se o nome de *passado causal de e*. Como consequência $e \notin C_e$.

Em resumo, entende-se formalmente uma *computação local* como o conjunto totalmente ordenado definido pelo par $(H_i, \rightarrow)$ e uma *computação distribuída* como sendo um conjunto parcialmente ordenado definido pelo par $(H, \rightarrow)$.

5.2.5 Faltas

Assume-se que os processos podem falhar por paragem, o que é modelado pela execução de um passo $paragem_i$. Neste caso, para todos os restantes passos e_i contidos na história de um processo p_i que falha, estão no passado causal dos passos de paragem, ou seja, $e_i \rightarrow paragem_i$, o que está de acordo com o facto de um processo que falha não enviar nem receber mais mensagens e de não recuperar. No entanto, assume-se que uma maioria de processos não falha.

[1] Como um mesmo passo e_i^k do algoritmo a_i pode ser executado mais do que uma vez na história de um processo, assume-se que as várias execuções podem ser distinguidos por um índice. Para maior clareza da notação, este detalhe é omitido, uma vez que é dedutível do contexto.

Admite-se ainda que tanto os processos como os canais podem falhar por omissão perdendo mensagens. No entanto considera-se que a probabilidade de uma mensagem não ser perdida é sempre superior a zero, ou seja, apenas se considera *perda justa*. Isto é equivalente a considerar apenas partições transitórias, pois caso contrário, qualquer problema interessante era trivialmente impossível.

Considera-se também a possibilidade de os canais duplicarem mensagens e embora não se assuma qualquer limite para o número de vezes que uma mensagem é recebida, assume-se que nenhuma mensagem é recebida um número infinito de vezes, ou seja, apenas se considera *duplicação justa*.

5.2.6 Consenso

A disponibilidade de um serviço de consenso é modelada pela existência dos eventos $propõe_i(k, v)$ e $decide_i(k, v)$, que correspondem à proposta e decisão de um valor v por um processo p_i para uma iteração k do problema do consenso. Assume-se ainda que nenhum processo propõe um valor para uma iteração sem ter decidido a iteração anterior[2] e que todos os processos que não falham executam um número infinito de iterações. Deste modo é verdade que para quaisquer processos p_i e p_j:

- se $decide_i(k, v) \in H_i$ e $decide_j(k, v') \in H_j$ então $v = v'$;
- se $decide_i(k, v) \in H_i$, existe um processo p_j tal que $propõe_j(k, v) \in H_j$;
- para todo o processo p_i, ou $decide_i(k, v) \in H_i$ ou então $paragem_i \in H_i$.
- para todo o processo p_i, $decide_i(k, v) \rightarrow propõe_i(k + 1, v')$;

5.2.7 Vista do grupo

Considera-se ainda que cada processo mantém associado a cada passo da sua história um conjunto de identificadores de processos que considera correctos. Este conjunto, $g_i \subseteq [1 \ldots n]$ é chamado a *vista do grupo* pelo processo p_i ou simplesmente a *vista* de p_i. Esta vista pode ser diferente para diferentes processos e em diferentes pontos do tempo em cada processo.

Não se faz no entanto qualquer pressuposto sobre precisão das vistas relativamente aos processos que de facto falham, sobre o modo como são actualizadas ou sobre os limites entre os quais podem variar. Assume-se apenas que cada passo na história de um processo tem como atributo uma vista, dizendo-se que ocorre nessa vista. Deste modo, para cada passo e_i^j da historia do processo, é conhecida a vista do processo nesse passo $g_i^j = \textsc{Vista}_p(e_i^j)$.

[2]Excepto na primeira iteração quando $k = 1$, em que a proposta é feita antes de qualquer decisão.

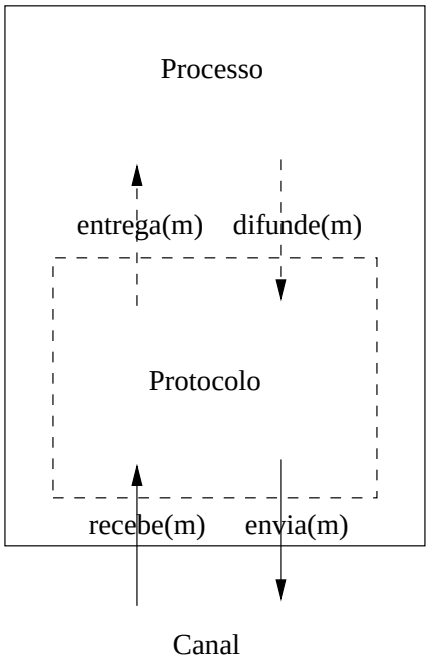


Figura 5.1: Protocolo no contexto de um processo.

5.3 Protocolos

5.3.1 Definição

Conceptualmente e de um modo informal, um *protocolo* é uma transformação de um canal de comunicação, segundo uma especificação de como uma troca de informação se deve processar para ser realizada com sucesso. Um exemplo de especificação de um canal é a difusão causalmente ordenada, que pode ser concretizada, por exemplo, por um protocolo correspondente a retenção de mensagens baseada em comparações de vectores de versões que são adicionados às mensagens [BSS91].

De um modo mais preciso, no modelo descrito podem considerar-se dois casos particulares de computação local, $difunde_p(m)$ e $entrega_p(m)$, que correspondam respectivamente ao envio e recepção de um mensagem m no canal resultante da transformação (Figura 5.1). Um protocolo identifica-se então no algoritmo executado por cada processo que relaciona estes eventos com os eventos $recebe_p(m)$ e $envia_p(m)$ associados ao canal de comunicação. Esse algoritmo é parte do algoritmo de cada processo e como tal propriedade da máquina de estados que descreve o processo.

5.3.2 Especificação

De acordo com a definição, a especificação de um protocolo de comunicação é uma restrição às execuções possíveis dos processos num sistema, impondo condições sobre os eventos de difusão e entrega contidos nas histórias de processos.

De um modo sucinto e sem perda de generalidade, podem classificar-se as mensagens e os processos em conjuntos, não necessariamente disjuntas, e impor restrições às histórias dos processos relativos às mensagens em cada uma dessas classes. A especificação do protocolo é dado pelo conjunto de todas as restrições relativas a cada uma das classes.

Deste modo, cada restrição k à história de processos é definida por:

- o conjunto $S_k \subseteq P$ dos processos onde é aplicável;
- o conjunto das mensagens $\{m : \rho_k(m, \dots)\}$ relevantes;
- a condição $\pi_k(H_i, m, \dots)$ imposta à historia dos processos $p_i \in S_k$;

sendo então a especificação de um protocolo Π dada pelo conjunto de todas as restrições:

$$\begin{aligned} \Pi = \{\forall p_j \in S_1, \rho_1(m, \dots) &\Rightarrow \pi_1(H_j, m, \dots), \\ \forall p_j \in S_2, \rho_2(m, \dots) &\Rightarrow \pi_2(H_j, m, \dots), \\ &\vdots \\ \forall p_j \in S_n, \rho_n(m, \dots) &\Rightarrow \pi_n(H_j, m, \dots)\} \end{aligned}$$

Uma especificação de protocolos é coerente se não for possível a partir dela deduzir ao mesmo tempo condições contraditórias sobre a história de um processo, ou seja, é falso que exista algum m e algum p_i tal que:

$$\Pi \vdash \pi_k(H_i, m, \dots) \wedge \neg\pi_k(H_i, m, \dots)$$

Apesar de ter que ser coerente, uma especificação não precisa de ser completa, ou seja, não é necessário que permita decidir sobre a veracidade de qualquer $\pi_k(H_i, m, \dots)$, uma vez que, é normal que a especificação deixe alguns graus de liberdade que podem ser decididos ou pelos algoritmos em particular ou aleatoriamente durante a execução.

Um algoritmo distribuído $A = (a_1, \dots, a_n)$ correspondente a um conjunto de n processos concretiza Π, $A \lll \Pi$, se e só se todas as histórias H que podem ser geradas por esse algoritmo cumprem as restrições de Π. É pois possível que um algoritmo distribuído concretize vários protocolos e que um protocolo seja concretizado por vários algoritmos distribuídos.

5.3.3 Classificação hierárquica

Uma classificação natural para protocolos é dada pela possibilidade de deduzir um conjunto de restrições a partir de outro. Deste modo, o conjunto de todos os protocolos tais que para quaisquer dois Π e Π':

$$\Pi \vdash \Pi' \wedge \Pi' \vdash \Pi$$

define uma classe $\|\Pi\|$ de especificações de protocolos equivalentes. Para todo $\Pi \in \|\Pi\|$ e $\Pi' \in \|\Pi'\|$ tal que:

$$\Pi \vdash \Pi'$$

então $\|\Pi\|$ é mais forte que $\|\Pi'\|$, definindo uma relação de ordenação parcial sobre classes de protocolos. Um exemplo é o facto de uma ordenação FIFO definida pela impossibilidade de duas mensagens serem entregues pela ordem inversa que foram difundidas, ou seja:

$$\forall p_j \in P, difunde_i(m) \xrightarrow{i} difunde_i(m') \Rightarrow \neg(entrega_j(m') \xrightarrow{j} entrega_j(m))$$

ser mais fraca que um ordenação causal, definida pela impossibilidade de qualquer processo entregar duas mensagens por ordem inversa à ordenação da sua difusão e entrega em qualquer processo, ou seja:

$$\forall p_k \in P, difunde_i(m) \rightarrow difunde_j(m') \Rightarrow \neg(entrega_k(m') \xrightarrow{k} entrega_k(m))$$

No entanto, ambas são incomparáveis com uma ordenação total [HT93] definida pela impossibilidade de qualquer processo entregar duas mensagens pela ordem inversa da sua entrega em qualquer outro:

$$\forall p_j \in P, entrega_i(m) \xrightarrow{i} entrega_i(m') \Rightarrow \neg(entrega_j(m') \xrightarrow{j} entrega_j(m))$$

5.3.4 Simetria

De acordo com a especificação dada, pode ainda fazer-se uma distinção entre protocolos simétricos, em que todas as restrições estão quantificadas universalmente nos processos, e protocolos assimétricos, em que diferentes restrições se aplicam a diferentes processos.

Uma consequência importante da simetria é que todos os processos podem executar o mesmo algoritmo, enquanto que protocolos assimétricos podem implicar diferentes algoritmos para diferentes processos.

Embora os protocolos de difusão em grupo sejam geralmente simétricos, em contraste com protocolos ponto-a-ponto, existem exemplos de protocolos assimétricos no contexto de grupos, como a difusão selectiva, onde alguns processos têm a garantia de receber as mensagens e os restantes a garantia oposta. Outro exemplo é o sistema PHOENIX[3] com os diferentes tipos de processos e consequentemente diferentes requisitos para os respectivos algoritmos.

[3] Ver Secção 3.3.4.

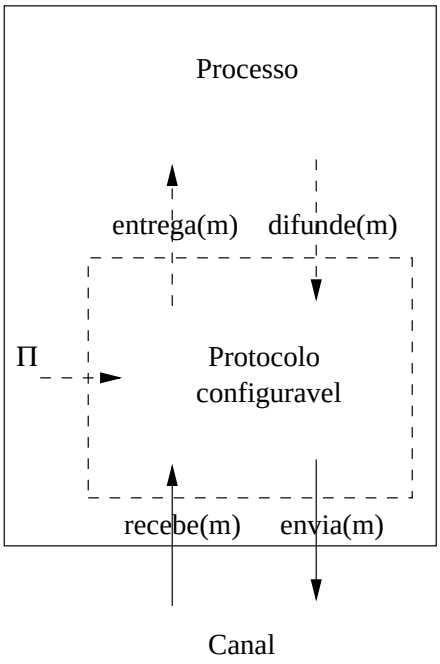


Figura 5.2: Protocolo configurável.

5.4 Protocolos configuráveis

5.4.1 Definição

É teoricamente possível imaginar a existência de um algoritmo genérico que em vez de cumprir uma especificação de protocolo fixa, seja capaz de cumprir uma especificação que lhe é passada como parâmetro, eventualmente, construída pelos próprios processos durante a sua execução. Para o efeito, basta que todas as restrições, relevantes para uma mensagem que é difundida ou recebida, sejam conhecidas pelo algoritmo antes de esses eventos acontecerem, ou em ultima análise, simultaneamente com esses eventos.

A estes algoritmos chama-se *algoritmos configuráveis* e à reunião de todos as especificações de protocolos que podem cumprir o seu *domínio*. Por outras palavras, considerar o domínio como o conjunto e todas as frases possíveis de uma linguagem e o algoritmo configurável um interpretador dessa linguagem.

Na prática, isto é possível apenas se as restrições a introduzir durante a execução seguirem um número restrito de formatos pré-definidos. Isto faz com que seja interessante explorar em que medida diferentes formatos de restrições se traduzem em diferentes problemas a resolver pelo algoritmo e em que medida se podem criar algoritmos especializados num pequeno número de formatos que possam ser compostos em algoritmos complexos que ofereçam um largo espectro de opções.

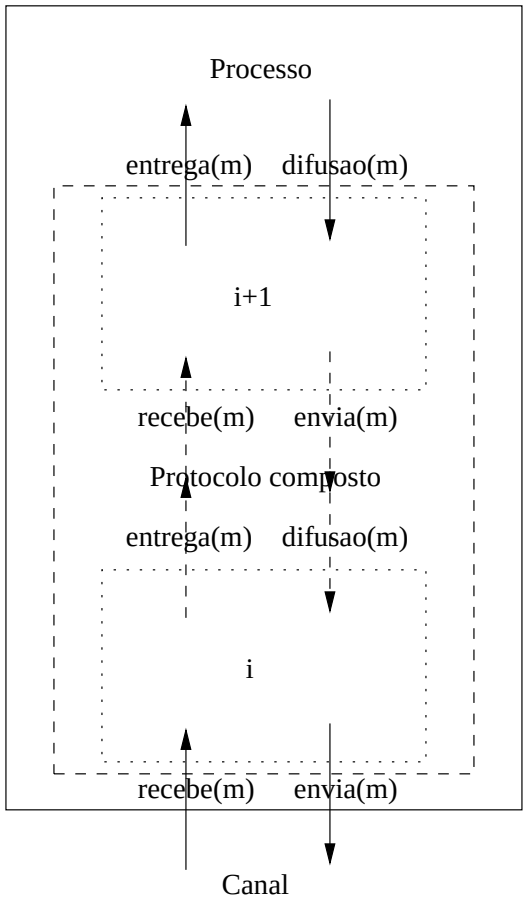


Figura 5.3: Composição de algoritmos.

5.4.2 Estratificação e composição

Uma vez que a concretização eficiente de um protocolo configurável é tanto mais provável quanto o domínio for restrito, é interessante considerar uma partição de um protocolo, por exemplo, segundo o formato das restrições, em sub-conjuntos disjuntos:

$$\Pi = \Pi^1 \cup \Pi^2 \cup \ldots \cup \Pi^n$$

e considerar algoritmos parciais $A^j \lll \Pi^j$.

Para cada dois algoritmos A^i e A^{i+1} definimos a sua composição $A^i \oplus A^{j+1}$, associativa mas não comutativa, como o algoritmo formado quando se faz equivaler em todo o processo p_i os eventos $difunde_i(m)$ e $entrega_i(m)$ de A^j a $envia_i(m)$ e $recebe_i(m)$ de A^{i+1} e se tomam os restantes como os eventos exteriores do algoritmo composto (Figura 5.3).

Assumindo que os algoritmos parciais não são contraditórios, ou seja, que A^k não contradiz A^j para quaisquer camadas $k > j$:

$$A^j \lll \Pi^j \Rightarrow (A^j \oplus A^k) \lll \Pi^j$$

ou seja, que o algoritmo composto concretiza a especificação da camada inferior então é verdade que concretiza a especificação de ambas as camadas:

$$(A^j \oplus A^{j+1}) \lll (\Pi^j \cup \Pi^{j+1})$$

e consequentemente, pela associatividade de $\oplus$, que:

$$(A^1 \oplus A^2 \oplus \ldots \oplus A^n) \lll \Pi$$

a composição de n algoritmos parciais concretiza a especificação completa.

Esta conclusão significa que um protocolo complexo pode ser especificado e concretizado como a composição de protocolos mais simples.

5.4.3 Acordo

Uma outra questão da maior importância no formato de uma restrição é quando os parâmetros adicionais de $\rho_i(m, \ldots)$ significarem que uma entrega a um processo depende não apenas da história do próprio processo mas também da história de outros processos:

$$\rho_i(m, H, \ldots) \Rightarrow \pi_i(H_j, m, \ldots)$$

Neste caso, podem distinguir-se duas situações em que a restrição depende:

- apenas de prefixos finitos e limitados das histórias dos processos, estando esse prefixos contidos no passado causal do passo que envia a mensagem;
- de prefixos não limitados e possivelmente infinitos das histórias de vários processos, ou seja, depende da futura evolução de outros processos.

No primeiro caso, na pior das hipóteses, é ainda teoricamente possível incluir nas mensagens os prefixos das histórias relevantes de modo a que cada destinatário possa avaliar inequivocamente a condição de entrega. Na prática, incluem-se apenas as secções relevantes das histórias dos processos codificadas de um modo eficiente, como por exemplo, sob a forma de vectores de números de sequência.

No segundo caso, a avaliação da condição significa que as histórias futuras de outros processos têm que ser ou conhecidas, esperando que ocorram os eventos relevantes, ou então condicionadas de modo a que não ocorram eventos que invalidem a decisão tomada. Para o efeito, é necessário chegar a

acordo com esses processos, o que implica o recurso ao consenso que integra o modelo.

A estes dois tipos de restrições pode ainda chamar-se respectivamente restrições de difusão-entrega (DE) e entrega-entrega (EE) devido ao facto de a entrega ser condicionada no primeiro caso apenas pela difusão e no segundo caso também pela entrega em terceiros.

5.4.4 Estratégias de concretização

Uma possibilidade de concretização de protocolos de comunicação configuráveis consiste em tomar o seu domínio como o conjunto de todas as frases de uma linguagem. Pode então definir-se uma gramática e um conjunto de palavras que geram essa linguagem e concretizar um protocolo configurável como um interpretador.

Como alternativa, pode considerar-se a partição da concretização em módulos, de modo que cada combinação possível de módulos corresponda a uma configuração possível do protocolo e consequentemente, todas as combinações possíveis correspondam ao domínio. Sendo um algoritmo uma máquina de estados formada por:

- um conjunto Σ de estados;
- um conjunto ϵ de estímulos;
- uma função de transição de estado $\delta : (\Sigma, \epsilon) \longrightarrow \Sigma$;
- e uma função de saída $\omega : (\Sigma, \epsilon) \longrightarrow \epsilon$;

então a modularização de um algoritmo passa necessariamente por uma decomposição da máquina de estados em módulos, o que pode ser feito segundo as seguintes estratégias:

- divisão do estado Σ em $\Sigma = \Sigma_1 \times ... \times \Sigma_n$, com a consequente divisão das funções δ e ω;
- divisão das funções segundo os estados Σ em funções parcelares para cada Σ_i tal que $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n$, para Σ_i disjuntos;
- divisão das funções segundo os estímulos ϵ em funções parcelares para cada ϵ_i tal que $\epsilon = \epsilon_1 \cup \ldots \cup \epsilon_n$, para ϵ_i disjuntos.

A primeira alternativa de modularização do algoritmo corresponde à composição de máquinas de estados independentes e como tal é a estratégia adequada para concretizar especificações estratificadas[4].

A segunda alternativa, permite em cada módulo especificar para um sub-conjunto de estados o efeito dos estímulos, pelo que é semelhante aos micro-protocolos baseados em eventos[5].

[4] Ver Secção 4.2.2.

[5] Ver Secção 4.2.3.

	Fiabilidade	*Ordem*	*Vista*
Difusão-Entrega	Sobreposição	Causal	Instantaneidade
Entrega-Entrega	Atomicidade	Total	Sincronismo

Tabela 5.1: Formatos de restrição.

A terceira alternativa permite para cada estímulo especificar as transformações de estados e saídas, pelo que se revela a mais indicada para para concretizar um algoritmo configurável, uma vez que faz corresponder um módulo do algoritmo a cada mensagem.

5.5 Parâmetros de configuração

5.5.1 Introdução

Tendo definido o conceito de protocolo configurável é necessário caracterizar quais os formatos de restrição relevantes para um conjunto tão vasto quanto possível de aplicações.

Para o efeito, considera-se uma estratificação em três classes, respectivamente, fiabilidade, ordem e diagnóstico. Cada uma destas é ainda repartida em função da necessidade de acordo dando origem a seis tipos de restrições resumidas na Tabela 5.1. A escolha destas categorias é em parte subjectiva, no entanto pode argumentar-se que:

- os diversos formatos de restrição são independentes, na medida em que nenhum deles pode de uma forma genérica ser reescrito nos outros;
- cada um dos tipos de restrição corresponde a um problema específico a resolver pelo algoritmo não existindo sobreposição entre eles;
- uma divisão em menos classes tornaria mais difícil a concretização das camadas do algoritmo;
- uma divisão em mais classes tornaria mais complexa a especificação, com a associada perda de clareza, sem benefícios em termos de correspondência com a concretização.

5.5.2 Fiabilidade

As restrições à liberdade dos algoritmos de entrega fiável cometerem erros são restrições à possibilidade de descartar mensagens recebidas sem as entregar. Deste modo, as restrições relativas à fiabilidade traduzem-se na

obrigação de entregar algumas mensagens e são da forma:

$$\rho_f(m, \dots) \Rightarrow \textsc{Entrega}_i(m)$$

em que a fórmula $\textsc{Entrega}_i(m)$ significa que em todas as execuções, a história de p_i, H_i, inclui uma e uma só vez o passo $entrega_i(m)$ ou então não o inclui nenhuma vez e inclui $paragem_i$, ou seja, p_i inevitavelmente entrega m uma vez ou falha, não a entregando nenhuma vez.

No que diz respeito à condição suficiente, ρ_f, podem ainda considerar-se dois formatos distintos, dependendo da necessidade de acordo ou não. No caso de não ser necessário acordo, pode resumir-se como:

$$\rho_{f,DE}(m, H_i) \equiv \nexists m' : m \in \textsc{Inc}(m') \wedge \textsc{Entrega}_i(m')$$

em que $\textsc{Inc}(m')$ é o conjunto de todas as mensagens cuja informação está incluída em m' e como tal não necessitam de ser entregues no caso de entrega de m'. Esta relação de sobreposição entre mensagens depende necessariamente da sua semântica e como tal a sua definição faz parte da definição da restrição.

Exemplos desta relação de sobreposição entre mensagens em sistemas existentes, que torna a entrega de algumas mensagens opcional, são os canais insistentes, as mensagens transparentes à causalidade e a implementação de ordem causal por aglomeração de mensagens, casos em que a sobreposição está contida na relação de causalidade. Outro exemplo são as mensagens de instalação de vistas, que se sobrepõem a todas as mensagens da vista anterior para todos os processo que entram na vista.

O problema a resolver pelo algoritmo para satisfazer as restrições deste formato é a retransmissão para eliminar perdas e a eliminação de duplicados, o que implica a manutenção da história das recepções em cada destino e uma aproximação dessa mesma história em cada origem.

Quando a condição suficiente implica acordo, então está relacionada com a coerência das mensagens entregues pelos vários processos:

$$\rho_{f,EE}(m, H) \equiv \exists p_j \in P : \textsc{Entrega}_j(m)$$

e que se traduz na atomicidade da difusão de uma mensagem. Existem também múltiplos exemplos deste tipo de restrição sempre que se considera a entrega atómica, necessariamente presente em qualquer sistema de difusão fiável.

O problema a resolver neste caso é o conhecimento da história de recepções por cada destino em todos os outros destinos, de modo a poder avaliar a condição de quórum sobre os processos de modo a não entregar nenhuma mensagem que não seja inevitavelmente entregue por uma maioria de processos, o que garante que pode ser retransmitida no caso de uma minoria de falhas de processos.

5.5.3 Ordem

As restrições à liberdade dos algoritmos de entrega ordenada cometerem erros são restrições à possibilidade de entregar pares de mensagens por um ordem errada. Deste modo, as restrições podem resumir-se a pares que não podem estar contidos na ordem de entrega:

$$\rho_o(m, m', \dots) \Rightarrow \neg \textsc{Entrega}_i(m' \rightarrow m)$$

em que a fórmula $\neg \textsc{Entrega}_i(m' \rightarrow m)$ significa em nenhuma execução correcta do sistema, nenhum prefixo da história de p_i, H_i, contém $entrega_i(m')$ sem conter também $entrega_i(m)$. Esta especificação pela negativa toma em consideração que a não entrega de qualquer uma das mensagens do par satisfaz a restrição de ordem o que não aconteceria se a restrição fosse pela positiva. Deste modo evita-se especificar implicitamente aspectos relativos à fiabilidade no sentido de tornar os diferentes formatos de restrição independentes.

No caso em que a condição suficiente, ρ_o, não depende de acordo então tipicamente reflecte uma relação de causalidade e é:

$$\rho_{o,DE}(m, m', H_i) \equiv m \in \textsc{Pred}(m')$$

A fórmula $\textsc{Pred}(m')$ representa o conjunto de mensagens que de acordo com o algoritmo são semanticamente predecessoras de m'. Deste modo define-se uma *causalidade semântica*, necessariamente um subconjunto da causalidade potencial $\rightarrow$. Isto permite que seja a aplicação a definir uma relação de ordenação mínima que permita uma maior concorrência.

O problema a resolver neste caso é o atraso de mensagens em função da história das entregas feitas localmente em cada destino, até que todas as suas predecessoras tenham também sido entregues.

Quando a condição suficiente depende de acordo então importa relacionar as entregas num processo com os restantes:

$$\rho_{o,EE}(m, m', H) \equiv \exists p_j \in P : \textsc{Entrega}_j(m \rightarrow m')$$

Nos sistemas existentes, este tipo de restrição está na base de protocolos de ordenação total quer entre mensagens ordinárias, quer entre estas e mensagens de mudança de vista[6].

Este caso implica a resolução do problema do consenso para decidir qual a ordenação de cada par que tem que ser entregue pela mesma ordem em todos os destinos. Visto de outra maneira, implica o acordo sobre sucessor de mensagens que não possam ser reordenadas.

[6]Também conhecidas como protocolos de esvaziamento do grupo.

5.5.4 Vista do grupo

As restrições à liberdade dos protocolos cometerem erros quanto à vista são restrições à vista em que uma mensagem pode ser entregue:

$$\rho_g(m, g, \dots) \Rightarrow \neg\textsc{Entrega}_i(m, g)$$

em que a fórmula $\neg\textsc{Entrega}_i(m, g)$ significa que em nenhuma execução correcta do sistema a história de um processo inclui o passo $e = entrega_i(m)$ tal que $\textsc{Vista}_i(e) = g$.

Tal como com os restantes formatos de restrição há a considerar dois casos. No primeiro caso relaciona a vista do emissor no momento da difusão com a vista no receptor no momento da entrega:

$$\rho_{g,DE}(m, g', H_i) \equiv \textsc{Difunde}_j(m, g')$$

e traduz-se na proibição de entregar mensagens em vistas incompatíveis com aquela em que foram difundidas. A definição de incompatível pode variar, sendo a mais comum, incompatível quando diferente. Intuitivamente, este tipo de restrição procura caracterizar uma entrega instantânea ideal, onde seria impossível a ocorrência de quaisquer outros eventos entre a difusão e a entrega da mensagem.

Para implementar restrições deste tipo, basta conhecer a vista aquando da difusão e o critério de incompatibilidade durante a entrega, pelo que se pode optar por duas estratégias:

- atrasar a mudança de vista até todas as mensagens nela difundidas e dela dependentes da vista terem sido entregues em todos os processos relevantes;
- mudar a vista implicitamente durante a entrega de cada mensagem de acordo com o especificado por cada mensagem.

Uma segunda forma relaciona a vista do grupo pelos processos que entregam a mesma mensagem:

$$\rho_{g,DE}(m, g', H) \equiv \textsc{Entrega}_j(m, g')$$

e traduz-se na impossibilidade de diferentes processos entregarem a mesma mensagem em vistas incompatíveis. Corresponde portanto, num sistema ideal, à noção de uma entrega síncrona a todos os destinatários, não admitindo outros eventos entre duas entregas da mesma mensagem.

Este tipo de restrição poderá também implicar acordo, dependendo do que for considerado incompatível, pelo que se pode optar por duas estratégias:

- atrasar a mudança de vista até todas as mensagens nela entregues em algum processo e dela dependentes terem sido entregues em todos os processos relevantes;

- mudar a vista implicitamente durante a entrega de cada mensagem de acordo com o acordado com os restantes processos, o que implica a utilização de um mecanismo de consenso.

5.5.5 Mudança de vista

Vale ainda a pena referir que esta especificação de mecanismos de gestão de vistas de grupos é incompleta quando comparada quer com serviços existentes quer com outras especificações. Existem duas razões fundamentais para que isto aconteça:

- outros serviços e especificações assumem a existência de uma mensagem de mudança de vista e concentram-se nas propriedades dessa mensagem relativamente às restantes;
- não é feita qualquer tentativa de especificar o que se entende por vivacidade.

A justificação para a primeira é que a correcção de algoritmos não depende, de facto, da existência de uma mensagem de mudança de vista, que é um artifício de concretização. Se necessário, como com qualquer outra mensagem, podem então especificar-se quais as restrições quer em termos de ordem como de fiabilidade de uma mensagem de mudança de vista, recorrendo aos formatos habituais para estas categorias. De facto, se for escolhida a opção de concretização que muda implicitamente a vista por cada mensagem entregue então não faz sequer sentido a existência de uma mensagem de mudança de vista.

A justificação para a não introdução de restrições quanto à vivacidade prende-se com o facto de serem subjectivas, ou seja, relativas a cada um dos processos e aplicações, o que faz com que não sejam genéricas. Por exemplo, um processo pode decidir iniciar uma expulsão de um membro de um grupo porque o suspeitou, porque fica sem mais espaço para armazenar mensagens para ele ou porque depende de uma mensagem que ele não consegue retransmitir para satisfazer uma dependência causal. Em qualquer dos casos tentará expulsá-lo, o que num sistema onde sejam impostas restrições como as descritas resultará na prática na necessidade de utilizar uma mensagem de mudança de vista que faça o esvaziamento do grupo e reúna acordo quanto à nova vista.

5.6 Conclusão

Neste capítulo apresenta-se um modelo para um sistema distribuído, que permite raciocinar sobre protocolos de comunicação. Um protocolo é definido como um conjunto de algoritmos que criam a abstracção de um canal de comunicação entre os respectivos processos.

Sendo um protocolo especificado por um conjunto restrições às histórias possíveis dos processos intervenientes, introduz-se o conceito de algoritmo configurável, capaz de interpretar uma especificação de protocolo, em vez de obedecer a uma especificação fixa.

A concretização eficiente de um algoritmo configurável passa pela definição de um conjunto restrito de formatos de restrição aceitáveis e pela definição de uma estratégia para as concretizar como módulos. Para o efeito considera-se restrições simétricas, com acordo e a estratificação como factores fundamentais para a caracterização de formatos de restrição e a modularização do algoritmo pela partição das funções de transição e saída segundo os estímulos.

Procura-se então caracterizar uma grande diversidade de serviços de difusão fiável recorrendo a seis formatos de restrição, resumidos na Tabela 5.1. A diferença fundamental em relação a outras especificações de protocolos de difusão é o facto de ser feita independentemente para cada mensagem e processo e não para todas as mensagens e processos de uma sessão.

Esta facto permite encarar a possibilidade de concretizar uma algoritmo configurável segundo estes parâmetros e que consequentemente possa ser utilizado para satisfazer uma larga gama de diferentes requisitos.

Capítulo 6

Concretização de protocolos configuráveis

6.1 Objectivos

De acordo com a definição, um protocolo configurável poderia ser concretizado como um interpretador de uma linguagem que permitisse descrever os diversos parâmetros segundo os quais se pretende modificar o protocolo. No entanto, uma concretização deste tipo seria certamente uma severa limitação ao desempenho dos protocolos.

Como alternativa, propõe-se uma decomposição de um protocolo em módulos correspondentes aos diversos parâmetros de especificação, que combinados possam formar as configurações de protocolos desejadas.

Embora a generalidade das arquitecturas existentes para o desenvolvimento de protocolos de comunicação tenham como objectivo comum a modularização tanto do projecto como da concretização, no sentido de obter protocolos reconfiguráveis, é importante identificar em que medida endereçam diferentes aspectos relacionados com a configuração de um protocolo.

Um primeiro aspecto é a adequação de uma infra-estrutura para construir protocolos complexos por composição de componentes mais simples, tais como camadas de protocolos ou micro-protocolos. Em alternativa, uma infra-estrutura pode suportar a modificação de um protocolo monolítico, por exemplo por herança ou parametrização.

Um outro aspecto é a possibilidade de configurar o comportamento de protocolos apenas em termos de sessões de comunicação, o que acontece nas pilhas de protocolos, ou de configurar diferentes aspectos enquanto decorre uma sessão, como é o caso dos mensageiros nas redes activas.

Finalmente, uma infra-estrutura pode servir apenas para configurar protocolos estáticos, criados antes do início da sessão de comunicação, ou permitir a evolução dos protocolos mesmo durante um sessão de comunicação.

Para concretizar protocolos configuráveis no sentido que foi definido no capítulo anterior, é necessário combinar estes três aspectos, para conseguir protocolos configuráveis pela composição de componentes, durante a execução da aplicação e cujas propriedades são definidas separadamente para cada mensagem.

A estes aspectos acresce ainda a dificuldade em projectar uma interface em o protocolo e a aplicação, uma vez que a especificação incremental da qualidade de serviço representa um fluxo de informação que não existe em protocolos não configuráveis.

Para o efeito desenvolveu-se uma infra-estrutura de suporte à concretização de protocolos que procura reunir os aspectos considerados positivos das várias abordagens existentes com uma concretização aberta de módulos de *software*.

6.2 Infra-estruturas de *software*

Os protocolos apresentados baseiam-se num conjunto de infra-estruturas de *software*, que oferecem componentes que podem ser combinados, especializados e melhorados para construir sistemas completos num determinado campo de aplicação [GHJV95].

Na base desta arquitectura está a *infra-estrutura de componentes* orientada por objectos, que através de um conjunto de regras para o programador e de uma biblioteca de classes define:

- o que é um componente de *software*;
- como podem componentes ser manipulados e combinados em estruturas mais complexas, independentemente da sua concretização e funcionalidade.

Como especialização deste infra-estrutura abstracta, a *infra-estrutura de protocolos* tem como objectivos:

- a abstracção de um protocolo de comunicação como um grafo de filtros de fluxos de mensagens, generalizando as arquitecturas de protocolos empilhados;
- fornecer um conjunto de componentes de manipulação de fluxos de mensagens, úteis para a generalidade de protocolos;
- permitir que as mensagens sejam activas, ou seja, que as mensagens sejam também componentes manipuladas pelos filtros apenas por intermédio de serviços;
- definir uma arquitectura de alto nível para redes activas, sob a forma de meta-objectos a associar a fluxos de mensagens, modificando o comportamento dos filtros;

- disponibilizar primitivas básicas, como passagem de mensagens não fiável e uma interface entre aplicações e protocolos de comunicação.

Finalmente, define-se uma *infra-estrutura de representação de histórias de processos*, que generaliza as operações com números de sequência, vectores e matrizes de versão, utilizadas por um grande conjunto de protocolos, nomeadamente, protocolos de difusão fiável.

6.3 Componentes

6.3.1 Definição

Um componente de *software* é uma unidade de composição que proporciona apenas serviços bem definidos e tem todas as dependências do contexto documentadas. Cada componente pode ser utilizado independentemente e está sujeito a ser manipulado por outros [SP97]. Ou por outras palavras, um componente é um módulo de *software* independente do contexto, tanto a nível conceptual como de concretização [CS97]. Para o efeito é fundamental a descrição objectiva dos requisitos de um componente, do mesmo modo que se descrevem os serviços por ele prestados [lB97].

Considerando os programas e os dados particionados e ocultados em *componentes*, a funcionalidade é percebida como um conjunto de *serviços* relacionados e cooperantes. Para desempenhar a sua função, um componente recorre a serviços prestados por outros componentes, que são especificados como as suas *dependências externas*.

Tanto os serviços exportados como as dependências são as *características* do componente, sendo o componente definido precisamente pelo conjunto das suas características. Qualquer característica é por sua vez sintácticamente descrita por um *tipo*, que é um conjunto de assinaturas de operações.

Um componente pode, se necessário, oferecer qualquer número de características. Isto significa que um mesmo componente pode exibir mais do que uma característica do mesmo tipo e que as características de um componente podem ser alteradas durante a execução do programa.

Sendo uma infra-estrutura orientada por objectos, os componentes podem ser especializados e aumentados por herança, pela redefinição e adição de características.

Uma dependência é satisfeita por um serviço de um tipo concordante com o seu, criando uma *ligação* entre os dois componentes. Dois tipos são concordantes se são idênticos ou se o tipo do serviço é uma extensão do tipo da dependência. A ligação reflecte uma relação cliente-servidor entre os dois componentes descrita pelo tipo da dependência. Além destas relações entre componentes, que estão devidamente documentados como características, os componentes são totalmente estanques, não guardando referências para quaisquer outros dados ou serviços externos.

6.3.2 Composição

Como consequência do número variável de características, cada componente pode ser ligado a um número também variável de outros componentes. Isto significa que um sistema complexo pode ser encarado como sendo um grafo onde os componentes e ligações são respectivamente os vértices e os arcos.

A definição da estrutura de grafos pode ela própria ser descrita por componentes, pois a normalização das relações entre componentes permite construir componentes de ordem superior que incluem e manipulam outros componentes e as suas inter-ligações. Como tal, grafos recorrentes num sistema complexo podem ser ocultados e reutilizados.

Um questão essencial para o sucesso desta estratégia é a possibilidade de exportar características seleccionadas dos componentes como características do próprio grafo.

Dependendo do componente em concreto usado para a composição, a estrutura do grafo pode ser definida de várias maneiras, por exemplo:

- conjuntos estáticos de componentes, suportando a criação de componentes por mistura[1];
- grafos estáticos, estruturados como um conjunto pré-definido de componentes e ligações, úteis para separar níveis de abstracção e esconder a complexidade;
- grafos dinâmicos, que criam e destroem componentes e ligações conforme necessário;
- grafos incompletos, que são completados com componentes disponibilizados durante a execução.

Componentes que permitem que a sua estrutura seja alterada dinamicamente, normalmente exportam essa possibilidade como um serviço. Isto permite que outros componentes manipulem a estrutura do grafo.

6.3.3 Especialização

De modo a especializar esta infra-estrutura genérica de componentes numa infra-estrutura para uma domínio de aplicação concreto, três tarefas têm que ser efectuadas:

i. definir um conjunto de especificações de tipos que capturem a sintaxe das relações cliente–servidor das entidades no domínio considerado;

ii. construir um conjunto de componentes, possivelmente abstractos, correspondentes às entidades identificadas;

[1] Do inglês *mix-in* [BC90].

iii. identificar grafos recorrentes ou diferentes níveis de abstracção e implementá-los como compostos.

A infra-estrutura resultante pode então ser usada pelo programador de aplicações, que a adapta às suas necessidades especializando os componentes pré-definidos ou utilizando-os para formar grafos, em conjunto com os componentes da aplicação. O processo pode ainda ser repetido sobre a infra-estrutura resultante, para a especializar para um sub-domínio de aplicação ainda mais concreto.

A utilização da infra-estrutura de componentes mais abstracta subjacente, encoraja ainda a utilização na mesma aplicação da diferentes infra-estruturas concretas em conjunto, para endereçar aspectos distintos do mesmo problema.

6.3.4 Concretização em Java

Para concretizar a infra-estrutura de componentes escolheu-se a linguagem JAVA [GJS96], devido a ser uma linguagem orientada por objectos que ao apresentar uma separação de hierarquias de tipos e de classes vai ao encontro dos requisitos da infra-estrutura de componentes, ao que acresce ainda:

- código binário compatível com uma grande variedade de plataformas, sobretudo se não for utilizada a componente gráfica da biblioteca de classes, indo de encontro à diversidade em sistemas distribuídos em grande escala [LY96b];
- disponibilidade de primitivas de comunicação básicas, como difusão selectiva [LY96a];
- disponibilidade de mecanismos de serialização automática de objectos e de uma representação externa normalizada [RWWB96];
- carregamento dinâmico de código executável e mecanismos de segurança, que em conjunto com a compatibilidade binária permite concretizar redes activas [LY96b];
- características que favorecem o desenvolvimento rápido de protótipos, aliadas à possibilidade de reutilização desses protótipos em produtos finais.

A concretização da estrutura de componentes na linguagem JAVA procura minimizar as diferenças da programação com a infra-estrutura de componentes relativamente à linguagem em si.

De acordo com este princípio, componentes simples são concretizados directamente como objectos. Os serviços por ele exportados são as interfaces que o objecto possui, sejam elas definidas implicitamente, herdadas ou

concretizadas explicitamente. Deste modo um cliente requisita um serviço simplesmente convertendo a referência que possui para o componente numa referência para o tipo do serviço desejado, invocando em seguida operações desse serviço.

As dependências externas são concretizadas como serviços de um tipo particular, que oferece operações para satisfazer a dependência fornecendo uma referência para uma concretização do serviço necessário.

A utilização de um interface uniforme para todas as dependências implica que não seja feita uma verificação estática da compatibilidade do serviço oferecido para satisfazer a dependência, o que tem que ser feito durante a execução. Por outro lado, a interface uniforme permite a criação de código genérico que manipula grafos arbitrários de componentes.

De acordo com a definição da linguagem JAVA, esta correspondência simples entre um componente e um objecto só pode ser satisfeita nas ocasiões em que:

- o número de características é fixo e definido aquando da compilação do componente;
- as diferentes características não podem ter métodos com a mesma assinatura e implementações diferentes;
- existe no máximo uma característica de cada tipo por componente e como consequência no máximo uma dependência externa.

Quando não estão reunidas estas condições um componente é então concretizado como um conjunto de objectos, formando um *componente dinâmico*. Estes são componentes simples que exportam um serviço suplementar que permite aos clientes obter referências para cada uma das características, que podem então ser concretizadas como objectos independentes. É então possível que um componente possua um número variável de características, com tipos possivelmente repetidos, sendo cada um deles concretizado por um objecto independente.

De modo a poderem ser distinguidas, as características de um componente dinâmico são identificadas por etiquetas. Estas etiquetas podem ser qualquer objecto válido em JAVA. São úteis, nomeadamente:

- o objecto de classe de cada característica, se não existe mais do que uma característica com o mesmo tipo;
- objectos de classe criados para o efeito, declarando novos tipos para cada uma das características;
- constantes declaradas como membros públicos da classe, se um número constante de características é desejado;

- quaisquer outros componentes, sobretudo se as características forem acrescentadas dinamicamente e relacionadas com outros componentes.

A uniformização do modo de utilização de componentes simples e dinâmicos é conseguida através da utilização de uma função de biblioteca que esconde as diferenças entre os diferentes componentes, utilizando classes como etiquetas nos componentes simples e objectos arbitrários em quaisquer outros. Deste modo, a única diferença entre um programa cliente de um objecto JAVA e um cliente de um componente deste infra-estrutura é a substituição da operação de verificação de tipos pela operação que obtém uma característica de um componente.

O Apêndice A apresenta alguns exemplos de programação com a infra-estrutura de componentes, com ênfase na composição de protocolos de comunicação.

6.4 Protocolos de comunicação

6.4.1 Protocolos como componentes

A infra-estrutura de protocolos assenta no princípio que um protocolo de comunicação é um grafo de filtros de um fluxo de mensagens, distribuído pelos vários processos intervenientes. Deste modo, generalizam-se os diversos modelos de protocolos em camadas.

Cada componente do grafo oferece um serviço por cada fluxo de mensagens que recebe e uma dependência por cada fluxo de dados que envia. O tipo das características de fluxo de mensagens resume-se a uma única operação, que aceita um objecto correspondente à mensagem como parâmetro.

A definição de um serviço de fluxo de mensagens é semelhante à definição de um `PushConsumer` no serviço de comunicação de eventos [Obj94] da norma CORBA [Obj93], em que a fonte "empurra" os dados. Em relação a uma definição em que os dados são "puxados" pelo destino, tem a vantagem de permitir associar uma actividade independente a cada mensagem, por oposição à associação de actividades a cada um dos componentes de protocolo.

Além das ligações de fluxo de mensagens, os componentes de protocolos podem ser ligados por outros tipos de características julgados convenientes, como por exemplo:

- uma condição, usado por exemplo por componentes que efectuam filtragem;
- uma tabela de correspondências, usada por exemplo para gerir várias sessões.

A existência de múltiplas características com tipo simples encoraja o desenvolvimento de componentes mais abstractos, que efectuam apenas uma função. Como consequência, cada componente de protocolo pode ser fragmentado em componentes mais simples, que resolvem problemas recorrentes em vários protocolos, tornando-os mais reutilizáveis em situações diversas.

Por outro lado, mesmo a reutilização de componentes complexos é simplificada pelo facto de se poder separar o que seria uma interface complexa num conjunto de serviços e dependências mais simples, que podem ser ligados em separado com outros componentes. Deste modo, o que seria uma dependência escondida ou uma operação a mais numa interface uniforme, torna-se numa ligação explícita e separadamente configurável entre componentes.

Com base nestes tipos de características é disponibilizada um biblioteca de componentes simples, úteis numa grande variedade de protocolos, como por exemplo:

- controladores de diversos dispositivos de comunicação de baixo nível, incluindo facilidades para serialização de objectos e fragmentação de pacotes;
- diversas interfaces entre um protocolo e as aplicações;
- componentes para modificação abstracta de um fluxo de dados, como armazenamento, multiplexagem, filtragem e duplicação;
- diversos tipos de *fábricas abstractas* [GHJV95], para modificação dinâmica de grafos de protocolos;
- componentes auxiliares para depuração, para simulação de redes e recolha de estatísticas.

Ao mesmo tempo disponibilizam-se componentes mais complexos que, por exemplo, concretizam padrões arquitecturais como a gestão de ligações, ou compostos por componentes mais simples, de modo a proporcionar abstracções úteis como a rede de objectos.

6.4.2 Mensagens como componentes

Além de servir para compor grafos de protocolos, a infra-estrutura de componentes permite também que as próprias mensagens que circulam nesse grafo possam ser ocultadas em componentes. Os protocolos manipulam as mensagens apenas através de serviços, o que permite usar composição de objectos para adicionar cabeçalhos às mensagens, em vez de recorrer à concatenação de sequências binárias.

Em concretizações tradicionais de protocolos, em que as mensagens são tratadas como sequências binárias, cada módulo é obrigado a manipular

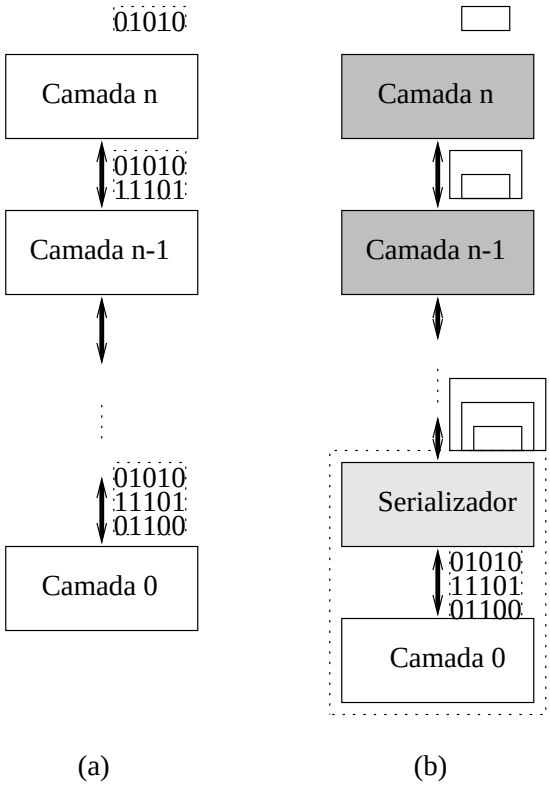


Figura 6.1: Comparação de protocolos de comunicação de (a) dados e (b) objectos. As duas camadas inferiores de (b) constituem uma rede de comunicação de objectos.

representações externas dos dados que acrescenta à informação em trânsito. Recorrendo à representação das mensagens como componentes é possível obter uma concretização em dois níveis, em que os protocolos mais complexos manipulam apenas componentes e apenas uma camada ao nível mais baixo é capaz de serializar os componentes produzidos pelas camadas superiores numa representação externa adequada (Figura 6.1).

Além das vantagens em termos de diminuição da complexidade do código dos protocolos de mais alto nível, esta arquitectura permite, em princípio, obter desempenhos tão bons ou mesmo superiores ao evitar cópias de informação, sem que para isso sejam necessárias quaisquer estruturas de dados adicionais para concatenar sequências binárias.

Embora para protocolos experimentais se possa utilizar uma qualquer biblioteca de serialização genérica, esta arquitectura não invalida a possibilidade de concretizar protocolos cuja representação externa esteja normalizada, o que pode ser conseguido através de uma camada de serialização especializada em que todas as representações externas de cabeçalhos legais na norma correspondam a objectos válidos e vice-versa.

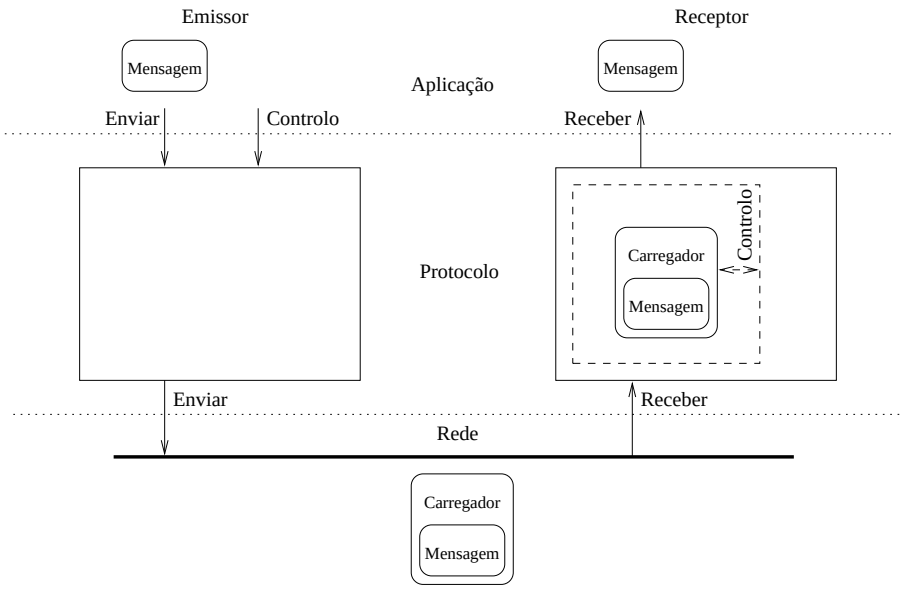


Figura 6.2: Uma mensagem como um componente activo: quando recebida pelo protocolo receptor, é executada modificando o estado deste.

6.4.3 Mensagens como componentes activos

Em qualquer protocolo não trivial, em que há vários tipos de mensagens trocadas, podem distinguir-se na concretização de um protocolo os procedimentos de manipulação das suas estruturas de dados dos procedimentos a executar quando da chegada de cada mensagem.

Normalmente todos estes procedimentos são concretizados como parte do mesmo módulo, ou seja, no componente de protocolo, enquanto os componentes correspondentes à mensagem são passivos e apenas disponibilizam serviços para consulta e alteração do seu estado.

Em alternativa, pode disponibilizar-se no componente de protocolo apenas os serviços para manipulação do seu estado e incluir nos componentes que constituem as mensagens, que passam a ser os componentes activos, os algoritmos correspondentes à transformação do estado do protocolo em resposta à recepção de cada mensagem (Figura 6.2).

A Figura 6.3 apresenta um exemplo de notação para especificar um componente de protocolo, ou *hospedeiro*, que se limita a ser uma enumeração dos serviços que exporta para os componentes *carregadores*, correspondentes às mensagens.

A especificação de um carregador consiste num conjunto de serviços que são tomados como parâmetros e num algoritmo executado no contexto dos serviços do hospedeiro, demonstrada na Figura 6.4. Além das construções algorítmicas normais consideram-se disponíveis as operações relacionadas

```
hospedeiro Exemplo exporta
 nome_1,   /* descrição do serviço 1 */
 nome_2,   /* descrição do serviço 2 */
           ⋮
 nome_n    /* descrição do serviço n */
fim
```

Figura 6.3: Exemplo da notação para um componente hospedeiro que disponibiliza vários serviços para os carregadores correspondentes.

```
carregador Exemplo para o hospedeiro Exemplo usa
 nome_1,   /* descrição do serviço 1 */
 nome_2,   /* descrição do serviço 2 */
           ⋮
 nome_n    /* descrição do serviço n */
faz
   linhas do algoritmo;
      ⋮
fim
```

Figura 6.4: Exemplo da notação para um componente carregador que transporta vários sub-componentes e executa um algoritmo no destino.

com entrega e envio de mensagens, enumeradas na Tabela 6.1.

A concretização de protocolos especificados como máquinas de estados é feita armazenando o estado como um serviço do hospedeiro (Figura 6.5) e concretizando cada parte da função de transição de estado correspondente a um estímulo no carregador correspondente (Figura 6.6).

Em relação a protocolos em que os procedimentos de recepção de mensagens fazem todos parte do mesmo módulo de *software*, a concretização de protocolos como um hospedeiro e um conjunto de carregadores, tem a vantagem de introduzir um nível adicional de modularização, permitindo modificar um protocolo incrementalmente pela modificação ou introdução de novos componentes correspondentes às mensagens, tal como é conseguido com as mensagens inteligentes.

De facto, esta estratégia de concretizar protocolos é semelhante a micro-protocolos baseados em eventos, na medida em que modulariza o projecto e concretização de uma máquina de estados. No entanto, neste caso a função de transição de estados é repartida segundo o estímulo e não segundo o estado. Como consequências este método:

- suporta transparentemente redes activas, como cápsulas e mensageiros, pela introdução a nível da rede de objectos do código dos componentes nas mensagens transmitidas que são pequenos programas autónomos;

`entrega` *objecto*	Entrega um objecto à camada superior.
`envia` *object*	Envia um objecto para a camada inferior.
`reenvia`	Reenvia o próprio carregador para a camada inferior.
`inicia` *objecto*	Inicia um outro carregador na mesma camada.

Tabela 6.1: Operações algorítmicas para descrição de carregadores.

```
hospedeiro Autómato exporta
  σ : Σ   /* estado */
fim
```

Figura 6.5: Exemplo da notação para um componente hospedeiro que concretiza uma máquina de estados simples.

- permite abrir as concretizações de protocolos aos programadores de aplicações.

6.4.4 Concretização aberta de protocolos

A concretização aberta de um módulo de *software* implica a disponibilização de uma meta-interface com o módulo, que permite a modificação do funcionamento do módulo. Num sistema orientado por objectos uma meta-interface é disponibilizada sob a forma de um meta-objecto, que é um objecto que descreve o funcionamento de outros objectos.

Considerando a concretização aberta de um protocolo de comunicação é necessário identificar quais as propriedades do funcionamento do protocolo que importa permitir ao programador de aplicação modificar. Essas propriedades são precisamente os invariantes que o protocolo impõe ao fluxo de mensagens que percorre o canal, como por exemplo a ordem das mensagens.

Essas propriedades são ainda determinadas pela informação que o protocolo acrescenta às mensagens aquando do seu envio, para serem cumpridas na entrega. Como consequência, podem considerar-se os cabeçalhos das mensagens, ou neste contexto, os carregadores, como sendo os meta-objectos que uma aplicação pode associar aos objectos a transmitir, para especificar as propriedades dessa transmissão.

A sessão de comunicação formada pela troca dos carregadores é assim uma meta-sessão de que descreve a sessão de comunicação constituída pela troca dos dados. Esta abordagem contrasta então com outras utilizações de reflexão, que ao considerarem o protocolo como um todo como o meta-objecto, restringem as possibilidades de configuração à selecção de um protocolo pré-definido (Figura 6.7).

A abertura da concretização de protocolos definidos deste modo implica

```
carregador Estímulo_i para o hospedeiro Autómato usa
  δ_i : Σ → Σ,   /* função de transformação do estado */
  ω_i : Σ → ϵ    /* função de saída */
faz
   σ ← δ_i(σ);
   envia ω_i(σ);
fim
```

Figura 6.6: Exemplo da notação para um componente carregador que concretiza a chegada de um estímulo a uma máquina de estados simples.

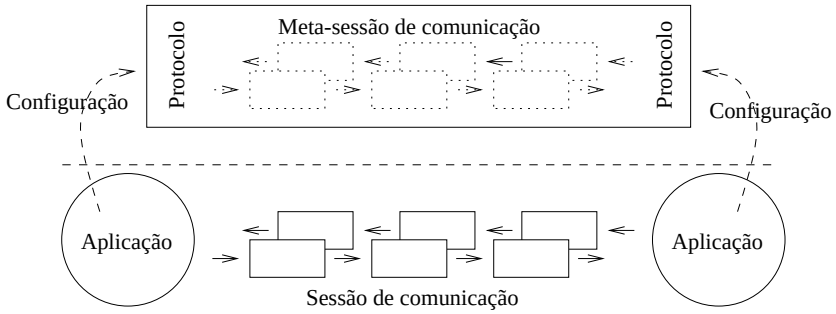
dar às aplicações a possibilidade de criar os cabeçalhos de cada mensagem (Figura 6.8), de modo a poder escolher a concretização para cada um deles de acordo com a sua semântica, em vez de esta escolha ser feita automaticamente pelo protocolo na ignorância desta informação. Para o efeito, uma aplicação cria um primeiro carregador, que é entregue ao protocolo emissor. Aí, esse primeiro carregador utiliza a informação disponibilizada pelo hospedeiro, além da informação disponibilizada pela aplicação, para criar um segundo carregador que prossegue até ao protocolo receptor.

No contexto desta infra-estrutura de protocolos a meta-interface proporcionada pelos protocolos pode ainda ser utilizada a vários níveis de abstracção:

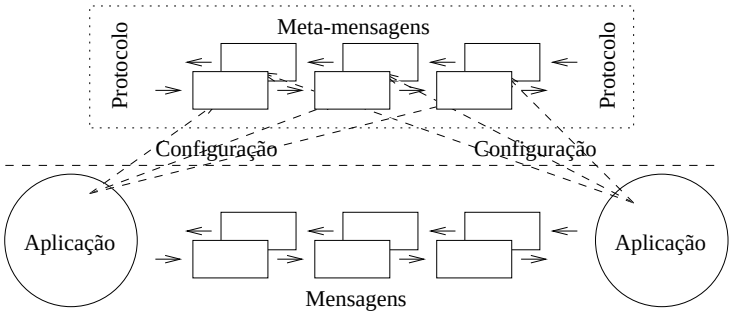
- ao nível do algoritmo do carregador, manipulando directamente os serviços do hospedeiro, o que proporciona a máxima flexibilidade mas implica o conhecimento de um interface mais complexa e do algoritmo a desempenhar por cada carregador;
- ao nível dos componentes utilizados como parâmetros para carregadores genéricos, o que embora não proporcione tanta flexibilidade é menos complexo, pois esse serviços aproximam-se mais das entidades emergentes modeladas;
- ao nível da selecção de um carregador, escolhido de uma biblioteca segundo a propriedade desejada.

Num sistema complexo, é conveniente estruturar o sistema nestes três níveis, garantido um espectro de opções adequado às diversas necessidades. Deste modo obtêm-se nos protocolos de comunicação as diversas vantagens reconhecidas às concretizações abertas, entre as quais:

- a clara separação entre interface e meta-interface simplifica a primeira reduzindo-a ao essencial, neste caso, duas operações para receber e enviar mensagens, tendo como parâmetro apenas o objecto que constitui a mensagem;



(a)



(b)

Figura 6.7: Reflexão em protocolos de comunicação: (a) protocolo como meta-objecto da sessão de comunicação; (b) cabeçalhos e mensagens de controlo como meta-objectos das mensagens.

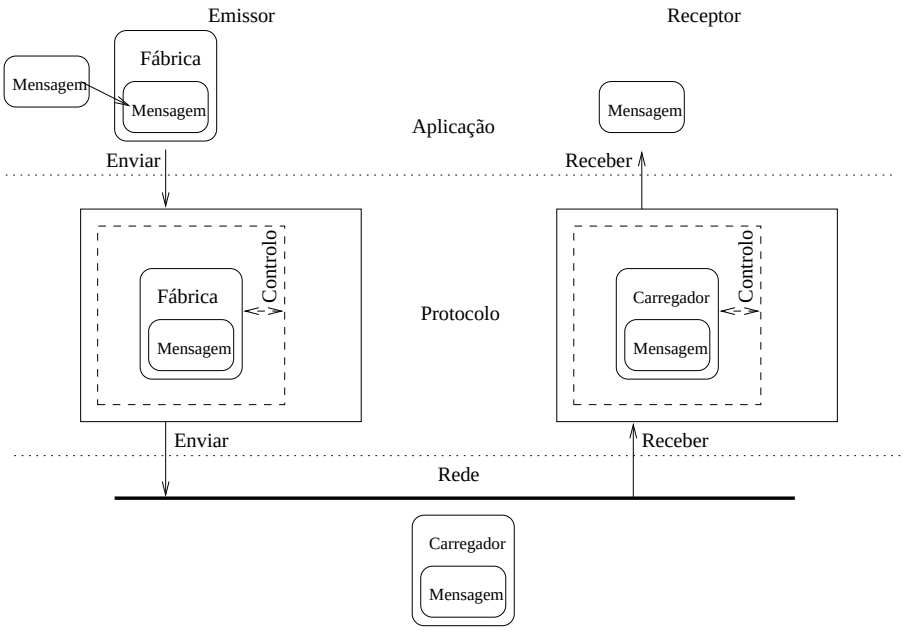


Figura 6.8: Protocolos abertos.

- possibilidade acrescida de adaptar o funcionamento de um módulo de *software*, tendo como consequência maior possibilidade de reutilização em diferentes situações;
- maiores possibilidades de a adaptação poder ser feita de um modo dinâmico, durante a execução do programa, por oposição a uma adaptação com base no mecanismo de herança ou em linguagens especializadas.

6.4.5 Estratificação por aspectos

Tal como acontece com protocolos tradicionais, a concretização aberta de protocolos pode ser utilizada para construir protocolos estratificados. A cada mensagem é associado um conjunto de carregadores aninhados, correspondendo cada um, a um dos níveis de hospedeiros. Quando recebida, em cada camada de hospedeiro o primeiro carregador é executado, libertando eventualmente a sua carga. Esta é um segundo carregador que prossegue para o nível seguinte e assim sucessivamente.

Para ser possível aninhar diferentes carregadores na origem de uma mensagem utilizam-se *carregadores-fábrica* aninhados pela aplicação no processo emissor. Cada carregador-fábrica é executado no hospedeiro correspondente no processo emissor durante o envio da mensagem e tem como função criar o carregador que irá ser executado no destino, sem alterar os carregadores

```
carregador Fábrica para o hospedeiro Exemplo usa
      C,   /* carregador para a camada inferior */
      d,   /* dados transportados */
 nome_1,   /* descrição do serviço 1 */
 nome_2,   /* descrição do serviço 2 */
           ⋮
 nome_n    /* descrição do serviço n */
faz
     ⋮
   envia C(Exemplo(d, ...));
fim
```

Figura 6.9: Exemplo de notação para um carregador-fábrica para o envio de uma mensagem num protocolo com várias camadas.

fábrica correspondentes às camadas inferiores nem os carregadores já criados pelas camadas superiores. Para o efeito é parametrizado com o carregador fábrica da camada inferior e com os carregadores das camadas superiores, que correspondem à mensagem (Figura 6.9).

No entanto, ao contrário do que acontece com protocolos tradicionais, cujas propriedades que oferecem são alteradas tanto em tipo como em concretização pela adição e substituição de camadas, a concretização aberta faz com que a adição de uma camada signifique apenas a adição de um grau de liberdade à configuração do protocolo, ao acrescentar um tipo de propriedade que pode ser modificado, sendo essa propriedade concretizada mais tarde.

Isto permite que uma pilha de protocolos seja mais estável entre diferentes aplicações, aumentado as possibilidades de inter-operação sem que para isso seja necessário a instalação remota de pilhas de protocolos, como é preconizado pelas redes activas.

6.5 Representação de histórias

6.5.1 Motivação

A concretização de protocolos de comunicação introduz a necessidade de avaliar condições sobre histórias de processos, tanto locais como remotos, nomeadamente quanto à recepção e entrega de mensagens. Para o efeito, têm que ser armazenadas representações tanto da história local como das histórias remotas relevantes.

Embora a história local possa ser conhecida com exactidão e armazenada de forma a tornar eficientes as operações de consulta relevantes, as

histórias remotas apenas podem ser conhecidas de uma forma aproximada, reunindo fragmentos recolhidos de mensagens trocadas para o efeito. Como consequência, as representações têm que ser adequadas também para serem enviadas como fragmentos facilmente reconciliáveis.

Para o efeito, é comum o recurso a números de sequência para representar eventos e prefixos de histórias, que em protocolos de grupo resultam em vectores de números de sequência. Por exemplo, um vector de números de sequência num protocolo de difusão causalmente ordenada representa ao mesmo tempo:

- a história de mensagens recebidas por um processo, quando armazenado no receptor;
- uma condição necessária sobre a história do receptor para a entrega da mensagem, quando armazenado nesta;

Ao mesmo tempo, o número de sequência relativo ao processo emissor incluído no mesmo vector quando isolado representa:

- a história de mensagens enviadas por um processo quando armazenado no emissor;
- o identificador único da mensagem quando em conjunto com uma identificação do emissor[2];

Ao utilizar a mesma estrutura de dados para diferentes objectivos optimiza-se quer o espaço ocupado, quer o tempo necessário ao seu processamento pelos algoritmos relevantes.

Porém, perde-se em clareza e sobretudo perde-se a possibilidade de manipular cada uma destas entidades em separado, com o objectivo de modificar o comportamento do protocolo. Por exemplo, num protocolo de difusão ordenada como o descrito perde-se a oportunidade de distinguir a condição de entrega da identificação da mensagem. Isto acontece porque a optimização das várias entidades numa mesma estrutura de dados simples, que é o vector de números de sequência, não é ocultada.

Uma alternativa para a representação de histórias que não requer este pressuposto é a enumeração explícita de eventos aos quais são atribuídos identificadores únicos [PBS89]. Deste modo, é possível oferecer operações mais de acordo com o conceito abstracto e portanto mais adequadas à possibilidade de adaptação a diferentes algoritmos. Porém, isto implica uma menor eficiência da utilização dos recursos, uma vez que a representação é menos compacta e é preciso um mecanismo suplementar para limitar o espaço de armazenamento necessário.

Em resumo, estão em conflito duas necessidades que se pretendem harmonizar:

[2]O que equivale a ser o identificador do passo em que o processo emissor envia a mensagem, uma vez que são univocamente correspondentes.

- oferecer tipos de dados abstractos que possam ser utilizados para diversas aplicações;
- optimizar esses tipos de dados tendo em conta alguns pressupostos quanto à sua utilização.

De modo a responder a estes requisitos, propõe-se uma representação de passos e histórias de processos sob a forma de componentes que escondam a sua estrutura interna, por forma a poderem ser utilizados em situações diversas. Os tipos e componentes resultantes formam a infra-estrutura de representação de histórias de processos.

6.5.2 Componentes

A infra-estrutura de representação de histórias permite representar identificadores de passos, fragmentos de histórias locais e fragmentos de histórias globais.

Cada passo é identificado como um número de sequência na história de um processo. Um fragmento de história local representa o conhecimento de um processo sobre os passos realizados por outros processos e como tal consiste em um ou mais identificadores de passos em conjunto com os identificadores dos processos aos quais se referem. Um fragmento de história global representa o conhecimento por parte de um processo dos fragmentos de histórias locais reunidos por outros e como tal é um conjunto de fragmentos de histórias locais em conjunto com os identificadores dos processos aos quais se referem. Relativamente à representação como números de sequência, estes três tipos de entidade correspondem respectivamente a números de sequência isolados, vectores e matrizes.

A cada um destes três tipos de entidades, correspondem dois tipos de serviços:

- um conjunto onde os fragmentos podem ser inseridos e que pode ser consultado quanto à inclusão de fragmentos;
- um fragmento que pode ser inserido no conjunto ou usado para o interrogar;

o que, para cada tipo, resulta em:

- um componente apresentando ambos os serviços;
- vários componentes que actuam apenas como fragmentos.

No que respeita à representação de passos, a infra-estrutura inclui os seguintes componentes:

Passo simples: Representa um único passo.

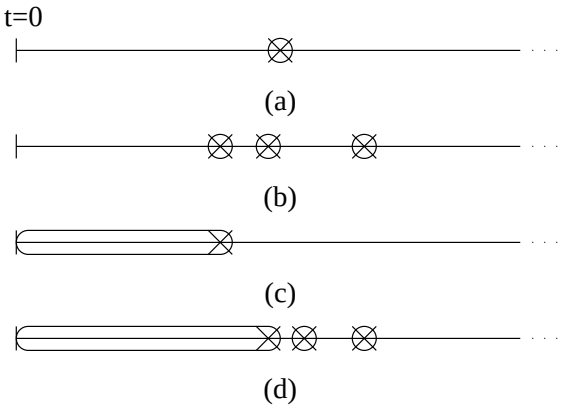


Figura 6.10: Representação de passos. (a) Um passo; (b) múltiplos passos; (c) sequência continua de passos; (d) conjunto de passos.

Múltiplos passos: Representa um conjunto qualquer de passos e optimizado para representar um conjunto pequeno de passos não necessariamente consecutivos.

Sequência de passos: Representa todos os passos de um processo até um determinado passo, sendo útil por exemplo, para representar passados causais.

Conjunto de passos: Representa um conjunto arbitrário de passos, reunindo num só componente a funcionalidade dos dois anteriores. É o único componente que pode ser modificado pela inclusão de novos passos e que pode ser interrogado.

No que respeita à representação de fragmentos de histórias locais, a infra-estrutura inclui os seguintes componentes:

Fragmento simples: Representa um único fragmento local, incluindo um identificador de processo e uma qualquer representação de passos.

Múltiplos fragmentos: Representa vários fragmentos locais, incluindo vários identificadores de processos e uma qualquer representação de passos.

Conjunto de fragmentos: Representa um conjunto arbitrário de fragmentos locais. É o único componente que pode ser modificado pela inclusão de novos fragmentos e que pode ser interrogado.

No que respeita à representação de fragmentos de histórias globais, a infra-estrutura inclui os seguintes componentes:

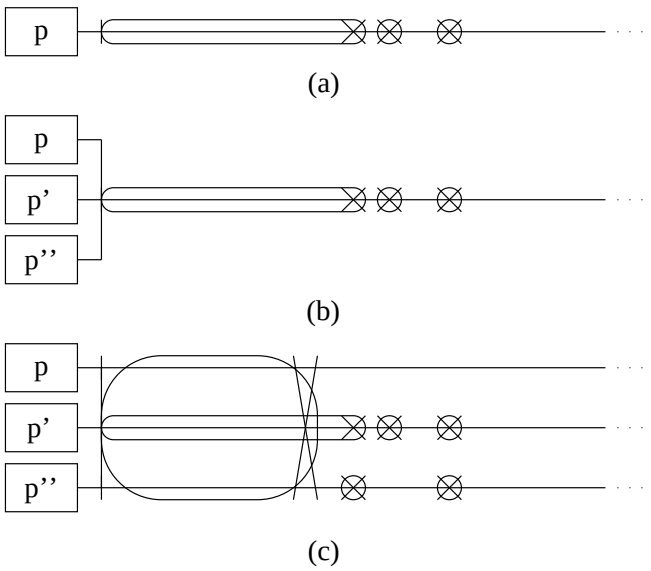


Figura 6.11: Representação da história de um grupo. (a) Especificação de histórias individuais mais um identificador de processo; (b) especificação de histórias individuais mais múltiplos identificadores de processos; (c) diferentes especificações de histórias para diferentes identificadores de processos, com optimização.

Fragmento simples: Representa um único fragmento global, incluindo um identificador de processo e uma qualquer representação de fragmentos locais.

Múltiplos fragmentos: Representa vários fragmentos globais, incluindo vários identificadores de processo e uma qualquer representação de fragmentos locais.

Conjunto de fragmentos: Representa um conjunto arbitrário de fragmentos globais. É o único componente que pode ser modificado pela inclusão de novos fragmentos globais e que pode ser interrogado.

Os conjuntos de fragmentos de histórias tanto locais como globais podem ainda ser sujeitos a uma optimização do armazenamento utilizado, estabelecendo um limite inferior para os passos representados (ver representação gráfica nas Figuras 6.10, 6.11 e 6.12). No entanto, isto impede que novos processos sejam acrescentados ao sistema sem que as histórias sejam sincronizadas, pelo que é apenas utilizável em alguns contextos. Neste diagramas as linhas horizontais representam a ordenação total das histórias e as caixas os fragmentos representados. O espaço de armazenamento ocupado por estas representações é proporcional ao número de cruzes dos diagramas.

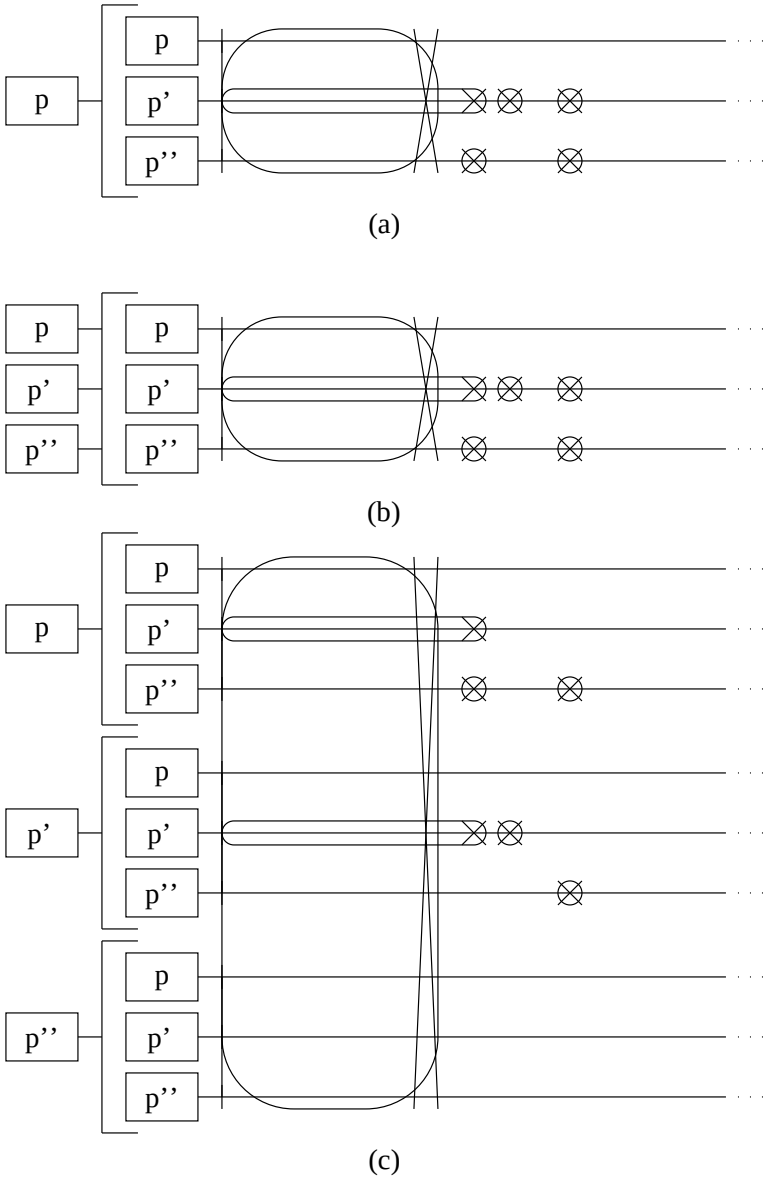


Figura 6.12: Representação da história de um processo. (a) Especificação de passos mais um identificador de processo; (b) especificação de passos mais múltiplos identificadores de processos; (c) diferentes especificações de passos para diferentes identificadores de processos com optimização.

6.5.3 Serviço de diagnóstico

A infra-estrutura de representação de histórias inclui ainda um terceiro tipo de serviço, que pode ser usado para extrair dos conjuntos de fragmentos de histórias, tanto locais como globais, informação sobre as actividades bloqueadas à espera de fragmentos. Nomeadamente, é possível saber quais os identificadores dos processos responsáveis pelo bloqueio de actividades.

Enquanto o serviço que permite bloquear actividades à espera da satisfação de uma condição de inclusão num conjunto de fragmentos permite concretizar as restrições de segurança de um sistema, por exemplo, ao testar as pré-condições numa entrega ordenada, este tipo de serviço permite testar condições relacionadas com a vivacidade de um sistema, por exemplo, ao identificar um processo que não retransmite mensagens e poderá portanto ter falhado. É portanto útil para identificar as situações em que é necessário efectuar uma mudança de vista num sistema de sincronismo virtual.

6.5.4 Representação de história futura

É ainda possível representar fragmentos de história futura, o que significa que são fragmentos de histórias locais que ainda não são conhecidos mas que como podem determinar o resultado de alguns protocolos são representados de modo a poderem ser manipulados.

Esta representação é opaca até ao momento em que é resolvida e se transforma em história passada. Até esse momento, dois fragmentos de história apenas podem ser comparados para averiguar se poderão vir a estar relacionados.

Cada fragmento é representado como um par (x, v), em que x é uma etiqueta, que pode ser qualquer objecto e v um valor lógico. Dois fragmentos de história estão relacionados se possuem a mesma etiqueta e em pelo menos um deles o valor lógico é verdadeiro.

A resolução de um fragmento de história futura transforma-o num fragmento de história local. Para o efeito, é necessário utilizar um serviço de resolução, que faça com que todos os fragmentos de história futura relacionados possam ser resolvidos de modo coerente. A Secção 7.4 ilustra a utilização de um protocolo de consenso, para resolver fragmentos de história futura usados em dependências de ordem para a entrega de mensagens, permitindo a ordenação de entregas relativamente a outras entregas.

6.6 Conclusão

Este capítulo apresenta um conjunto de infra-estruturas para desenvolvimento de protocolos configuráveis. A infra-estrutura mais básica define o que são componentes de *software* e como podem ser combinados.

A infra-estrutura de protocolos baseia-se na abstracção de protocolos de comunicação como filtros de um fluxo de mensagens e define um conjunto de componentes úteis para uma variedade de protocolos.

A infra-estrutura de protocolo encoraja ainda numa primeira etapa, a concretização das próprias mensagens como componentes que os respectivos protocolos manipulam através de um conjunto de serviços, e numa segunda etapa, a concretização das mensagens como componentes activos que modificam o estado do protocolo através de um conjunto de serviços.

Pela abertura da concretização de protocolos, dando à aplicação a possibilidade de seleccionar quais o componentes associados a cada mensagem, torna-se possível a concretização de protocolos configuráveis.

A concretização aberta é a característica que permite que a especificação incremental da qualidade de serviço seja feita através de uma interface simples e modular.

Finalmente apresenta-se a infra-estrutura de representação de histórias, que ao oferecer componentes necessários para a concretização da maioria de protocolos de difusão torna possível a concretização em separado e interoperação de componentes carregadores diversos.

Capítulo 7

Protocolos configuráveis de difusão fiável

7.1 Objectivos

Recorrendo às infra-estruturas de componentes, protocolos e representação de histórias, pode agora apresentar-se uma concretização de protocolos configuráveis de difusão fiável. Esta concretização efectuada na plataforma JAVA resulta na infra-estrutura de comunicação fiável GROUPZ.

Os objectivo principal do GROUPZ é ser uma concretização de referência da especificação apresentada que:

- mostre que a especificação de um protocolo configurável pode ser concretizada;
- mostre em que medida as infra-estruturas desenvolvidas suportam essa concretização;
- utilize algoritmos simples, que apesar da complexidade introduzida pela possibilidade de configuração permitam:
  - uma avaliação objectiva da sua correcção;
  - uma mais profunda compreensão das possibilidades de configuração por parte dos programadores de aplicações;
- disponibilize uma plataforma para prototipagem rápida de aplicações que tirem partido da configurabilidade, de modo a que possam ser tiradas conclusões a incorporar em futuras versões.

Embora o suporte a redes em grande escala seja a motivação inicial para este trabalho, esta concretização não pretende ser uma solução global para o problema incorporando todas as características desejáveis. Por exemplo, não

Figura 7.1: Estrutura de uma pilha de protocolos completamente configurada.

é considerada a possibilidade de partição em sub-grupos e a sua posterior junção.

Também não é objectivo desta proposta de concretização de um protocolo configurável de difusão fiável utilizar os algoritmos mais eficientes conhecidos para cada um dos problemas que é necessário resolver. Por exemplo, é utilizado um simples protocolo de consenso para efectuar a ordenação total de mensagens.

Esta concretização é feita em três camadas de componentes hospedeiros, em conjunto com os respectivos carregadores, sobre um componente de difusão de objectos (Figura 7.1).

O componente de difusão de objectos efectua duas tarefas principais:

- serialização, ou seja, conversão dos objectos de e para um representação externa apropriada a ser transmitida;
- transmissão das mensagens resultantes da serialização através dos mecanismos de comunicação disponíveis.

Além disso, este componente proporciona um mecanismo para o diagnostico aproximado de falhas de processos.

A camada de entrega fiável permite especificar as restrições relativas à entrega de mensagens. Para o efeito tem que endereçar os problemas relacionados com perda e duplicação de mensagens no canal de comunicação básico e o acordo quanto à recepção de mensagens para prevenir incoerência na presença de falhas.

A estratégia genérica para recuperação de perdas consiste na retransmissão das mensagens. A duplicação é evitada armazenando uma represen-

tação dos identificadores das mensagens recebidas anteriormente e a atomicidade é conseguida recorrendo à reunião de um quórum de recepção antes da entrega.

A camada de ordenação de mensagens permite especificar restrições relativas à ordem de entrega das mensagens em cada um dos destinatários. Para o efeito, retém as mensagens em trânsito até que todas as outras mensagens enumeradas como dependências de ordem tenham sido entregues.

Em consequência da utilização de fragmentos de história futura para especificar restrições de entrega em relação a outras entregas concorrentes, esta camada tem que ser capaz de resolver histórias futuras em histórias locais, o que é feito recorrendo a um protocolo de consenso. O algoritmo de consenso é também concretizado recorrendo a um hospedeiro e a um conjunto de carregadores[1].

A camada de gestão de vistas permite à aplicação fazer uma gestão coordenada da composição do grupo, impondo restrições à vista em que cada mensagem é entregue. Esta camada serve ainda como exemplo das vantagens que uma aplicação complexa pode retirar da configurabilidade de um protocolo. Neste caso, da configurabilidade das camadas de fiabilidade e ordenação, cuja funcionalidade normal é suficiente para concretizar o mecanismo de gestão de vistas sem recurso a operações especiais.

7.2 Difusão de objectos

7.2.1 Serialização e fragmentação

Como a abstracção de uma rede de comunicação de objectos é construída sobre uma rede de transmissão de dados sob a forma de sequências binárias de comprimento limitado, é necessário converter os objectos para uma representação adequada a ser transmitida, por um processo de serialização e fragmentação, a partir da qual podem ser reconstituídos depois de recebidos.

Embora possam ser concebidos componentes especializados de serialização para protocolos específicos, nesta primeira versão do GROUPZ utiliza-se apenas a serialização de grafos de objectos da plataforma JAVA [RWWB96, Sun96a].

Fazendo parte integrante da especificação da linguagem, a serialização de objectos JAVA é totalmente transparente e automática, quer para o programador de uma classe como para os seus utilizadores, na medida em que não precisa de ser escrito qualquer código adicional para descrever as estruturas de dados.

Embora esta decisão tenha vantagens significativas em termos de rapidez de desenvolvimento, produz representações pouco compactas e como tal pouco adequadas a protocolos de alto desempenho ou cuja representação

[1] Esta concretização é apresentada em detalhe no Apêndice C.

externa esteja normalizada. A sua utilização justifica-se numa situação de prototipagem rápida de protocolos, que quando estabilizam podem ser optimizados pela concretização da uma camada de serialização especializada.

O componente de serialização genérico depende de um serviço de fluxo de caracteres como destino para a representação externa produzida e como origem quando efectua o procedimento inverso. Para o efeito pode ser usado um fluxo de memória JAVA criando um vector de caracteres para cada mensagem.

Em alternativa, quando a representação externa de cada objecto é demasiado extensa para o dispositivo de difusão utilizado, o GROUPZ disponibiliza um fluxo de caracteres que produz directamente uma representação fragmentada durante o envio e que reúne fragmentos durante a recepção, permitindo que a operação de fragmentação se realize sem nenhuma cópia adicional. O mesmo componente é capaz de reagrupar e reordenar os fragmentos, para reconstituição da mensagem. Não é no entanto feito qualquer esforço para retransmitir fragmentos perdidos, casos em que toda a mensagem é descartada.

7.2.2 Difusão de unidades de dados

Ao nível mais baixo, as unidades de dados produzidas pelos componentes de serialização e fragmentação têm que ser encaminhadas através de componentes controladores correspondentes aos mecanismos básicos de comunicação.

Existem no GROUPZ três mecanismos básicos de comunicação:

- componente controlador para passagem de mensagens ponto-a-ponto;
- difusão IP utilizando um endereço de difusão com o componente de difusão ponto-a-ponto;
- difusão selectiva IP [Dee88] utilizando um endereço de difusão selectiva e um componente controlador apropriado.

Destes o preferido é sempre a difusão selectiva, visto ser o mais eficiente em termos de largura de banda da rede e de tempo de processamento das máquinas que não participam na comunicação. A difusão pode ser útil em casos em que a difusão selectiva não esteja disponível. O envio de mensagens ponto-a-ponto pode ser utilizado para:

- interligar redes de difusão;
- simular difusão selectiva.

A simulação de difusão selectiva é feita mantendo em cada processo uma lista aproximada, por excesso, dos endereços dos restantes processos restantes e repetindo cada uma das mensagens para todos eles.

Neste caso, a lista é mantida recorrendo a um número arbitrário e possivelmente variável de servidores, de moda a tolerar faltas. Para se juntar ao grupo informal de difusão, um processo envia regularmente o seu endereço para um ou mais servidores, onde é acrescentado à lista que passa a receber sempre que actualizada. A manutenção da lista reutiliza os componentes desenvolvidos para a detecção de falhas de processos.

7.2.3 Suspeita de falhas

A suspeita de falhas de processos pode ser obtida através da observação da *pulsação* de cada processo ou da emissão de um *desafio*. Um desafio consiste em interrogar cada processo do qual não seja observada nenhuma mensagem durante um intervalo de tempo. Um processo que não responda ao desafio depois de um determinado período de tempo é suspeito.

No GROUPZ é utilizada a pulsação, que é um fluxo de mensagens produzidas por cada processo a intervalos regulares. Um processo é suspeito por outro se a sua pulsação não é observada durante um determinado período de tempo. O processo deixa de ser suspeito quando a sua pulsação é recebida de novo.

Em ambos os casos, tanto a pulsação como os desafios e respectivas respostas podem ser acrescentados às mensagens normais, pelo que são sensivelmente equivalentes em termos de recursos consumidos.

Durante a observação da pulsação, a determinação do período de tempo a partir do qual se suspeita um processo pode ser feita de duas maneiras:

- um período constante, estritamente maior que o período da pulsação;
- um período variável, resultante de uma adaptação ao período de pulsação observado de cada processo.

Embora a programação de um período constante seja mais simples, a sua utilidade prática é reduzida, uma vez que perssupõe sistema o facto de diferentes processos concordarem no intervalo de tempo constante entre mensagens.

Um período de tempo adaptável assume apenas uma probabilidade de o desvio entre dois intervalos consecutivos ser limitado, o que é um pressuposto bastante mais realista. O método de previsão utilizado no GROUPZ utiliza uma média móvel para estimar o tempo entre cada duas mensagens e é parametrizado com uma estimativa inicial, o desvio percentual da estimativa que se admite provocar uma suspeita e a velocidade de adaptação da média. Deste modo, assegura-se que:

- todo o processo que falhe deixa de emitir a pulsação e inevitavelmente todos os processos correctos o suspeitam por o tempo depois da chegada da última mensagem de pulsação exceder o limite[2];

[2]A existência de uma última mensagem numa rede onde se admite duplicação das mensagens é válido pois a duplicação é sempre finita.

- admitindo que há um processo cuja pulsação a partir de um tempo é recebida com intervalos que variam menos que um desvio percentual por todos os outros processos, então a média estimada move-se para a média da pulsação recebida e o processo nunca mais é suspeito.

Em termos práticos, estes pressupostos podem ainda ser mais fracos, pois como basta assumir que a não suspeita de um processo correcto se mantém apenas pelo tempo suficiente para concluir a execução de um algoritmo, então basta assumir que há um processo cuja pulsação é recebida com intervalos que variam menos que um desvio percentual por todos os outros processos, apenas durante um intervalo de tempo suficientemente longo para a terminação do algoritmo em questão.

7.3 Entrega fiável

7.3.1 Algoritmos

A fiabilidade de entrega de mensagens implica a inevitabilidade da entrega de mensagens que pela especificação não podem ser perdidas, quer por não haver substituição ou pela sua entrega noutros processos ser inevitável, tal como especificado na Secção 5.5.2.

Em ambos os casos, a entrega inevitável sobre canais que podem perder mensagens implica a sua retransmissão até que a recepção aconteça e seja do conhecimento do emissor. A transmissão deste conhecimento pode ser feito recorrendo a algoritmos baseados em confirmação positiva ou confirmação negativa [PBS89].

Num algoritmo baseado em confirmação positiva, o receptor confirma ao emissor a recepção de cada mensagem. Até que essa confirmação seja recebida, o emissor deve retransmitir a mensagem. Quando recebida a confirmação, pode ser libertado o espaço de armazenamento associado à mensagem. O período de tempo entre cada duas retransmissões pode ser:

- constante;
- um valor exponencialmente crescente mas limitado;

Embora o primeiro seja mais simples de programar, pode ser problemático em situações de perda de mensagens por sobrecarga da rede, caso em que a repetição das transmissões vai contribuir para essa sobrecarga. No entanto, se o intervalo for suficientemente grande para não sobrecarregar a rede, introduz um atraso proporcionalmente grande em qualquer mensagem que tenha que ser retransmitida. A utilização do período variável resolve ambos estes problemas.

Um algoritmo baseado em confirmação negativa implica que o receptor confirme ao emissor a não recepção de uma mensagem, caso em que deve ser retransmitida. Este tipo de algoritmo implica que:

```
hospedeiro Fiável exporta
  H_g,   /* conjunto de fragmentos de história global */
   f_g   /* fábrica de fragmentos de história global */
fim
```

Figura 7.2: Componente hospedeiro para entrega fiável.

- se estabeleça um limite de tempo medido pelo receptor a partir do qual uma mensagem é considerada perdida;
- um mecanismo suplementar para o receptor saber quais as mensagens que devia ter recebido:
  - as mensagens têm referências para as mensagens previamente emitidas pelo mesmo processo e ainda não confirmadas;
  - cada receptor interroga a intervalos regulares os emissores dos quais não tenha recebido qualquer mensagem da qual possa retirar as referências para mensagens perdidas;
  - em protocolos que impõem uma ordenação às mensagens, as referências para ordenação podem ser reutilizadas como referências para confirmação negativa.

É ainda necessário que sejam emitidas pelos receptores confirmações positivas da recepção, de modo que o espaço de armazenamento para mensagens em cada emissor possa ser recuperado.

O GROUPZ utiliza um sistema baseado em confirmação positiva com intervalos de tempo exponencialmente crescentes, o que conjuga a simplicidade do algoritmo com um desempenho razoável, quer em termos de mensagens transmitidas como de atraso em caso de retransmissão.

7.3.2 Hospedeiro

O componente hospedeiro correspondente à camada de difusão fiável apresenta dois serviços internos (Figura 7.2):

- um conjunto H_g de fragmentos da história global de mensagens recebidas pelos vários processos;
- uma fábrica f_g de fragmentos de história global.

O primeiro permite saber exactamente quais as mensagens já recebidas pelo processo local e inevitavelmente quais as recebidas pelos restantes. Este conhecimento é aproximado porque embora se possa concluir que uma mensagem foi recebida por um outro processo, nunca de pode concluir o contrário. No entanto, esta informação é suficiente para avaliar condições referentes a:

```
carregador Difusão para o hospedeiro Fiável usa
    C,   /* carregador para a camada inferior */
    d,   /* dados transportados */
   id,   /* fragmento de história local (identificação) */
  ent,   /* fragmento de história global (condição de entrega) */
 term,   /* fragmento de história global (condição de terminação) */
 conf    /* fragmento de história global (outras confirmações) */
faz
   envia C(Entrega(d, id, conf, ent, term, conf));
fim
```

Figura 7.3: Componente carregador genérico para difusão fiável.

- eliminação de mensagens duplicadas no processo local;
- armazenamento e retransmissão de mensagens até terem sido recebidas por todos os processos remotos relevantes;
- avaliação de condições de quórum de recepção para entrega de mensagens.

Este componente é actualizado por fragmentos de história global contidos nos carregadores trocados, sendo a correcção desse processo da responsabilidade dos carregadores.

Para o efeito, esses fragmentos são obtidos a partir da fábrica de fragmentos de história global e dos identificadores das mensagens, que são fragmentos de história local. A fábrica de fragmentos de história global permite transformar fragmentos de história local, correspondentes aos eventos de envio de mensagens, em fragmentos de história global correspondentes à sua recepção pelo processo, ou conjunto de processos, representados pelo hospedeiro.

A utilização de uma fábrica para transformar os identificadores das mensagens nos fragmentos de história correspondentes à sua recepção permite que os carregadores não dependam da representação das histórias e ainda que um hospedeiro possa representar mais do que um processo. Esta última característica é fundamental para a interligação de sub-grupos utilizando representantes locais dos membros remotos.

7.3.3 Carregadores

O carregador genérico para difusão fiável (Figura 7.3) limita-se a difundir o carregador de entrega para todas os processos, incluindo para si próprio, terminando em seguida. Como um canal de um processo correcto para si próprio é garantidamente fiável, a retransmissão é feita a partir dos carregadores de entrega de modo a poder garantir as restrições de entrega.

```
carregador Entrega para o hospedeiro Fiável usa
     d,   /* dados transportados */
    id,   /* fragmento de história local (identificação) */
   ent,   /* fragmento de história global (condição de entrega) */
  term,   /* fragmento de história global (condição de terminação) */
  conf    /* fragmento de história global (outras confirmações) */
faz
1     H_g ← H_g ∪ conf;
2     t ← f_g(id);
3     envia Confirmação(t);
4     se t ⊈ H_g então
5        terminado ← F;
6        actividades
7           ||entrega:
8              se t ⊆ ent então
9                 H_g ← H_g ∪ t;
10                espera por ent ⊆ H_g;
11                entrega d;
12             fim
13             espera por term ⊆ H_g;
14             terminado ← V;
15          ||retransmite:
16             ΔT ← T_inicial;
17             enquanto ¬terminado faz
18                espera por ΔT;
19                ΔT ← max(ΔT * 2, T_max);
20                reenvia;
21             fim
22       fim
23    fim
fim
```

Figura 7.4: Componente carregador genérico para entrega fiável.

```
carregador Confirmação para o hospedeiro Fiável usa
  conf   /* fragmento de história global */
faz
   H_g ← H_g ∪ conf;
fim
```

Figura 7.5: Componente carregador de confirmação para entrega fiável.

O carregador genérico para entrega fiável (Figura 7.4) é baseado no método de confirmação positiva, executando:

- a entrega de uma confirmação de recepção, se necessário (linha 1);
- usa o hospedeiro para transformar o fragmento de história local correspondente à sua identificação num fragmento de história global correspondente à recepção da mensagem em causa (linha 1), com o que:
  - difunde um carregador auxiliar para comunicar esse fragmento de história (linha 3);
  - verifica se a mensagem transportada ainda não foi localmente recebida e se a mensagem é para ser entregue localmente, permitindo eliminação de duplicados e difusão selectiva (linhas 4 e 8);
  - actualiza a história do hospedeiro (linha 9);

Inicia então uma segunda actividade concorrente em que retransmite periodicamente a mensagem (linhas 15 a 21) enquanto:

- espera por uma condição de entrega arbitrária sobre a história global de recepções tal como conhecida pelo hospedeiro local, permitindo a reunião de um quórum prévio à entrega (linha 10);
- entrega os dados transportados (linha 11);
- espera por uma outra condição arbitrária sobre a história global de recepções para terminar a retransmissão periódica (linha 13).

O carregador de confirmação (Figura 7.5), limita-se a entregar a confirmação positiva da recepção de uma mensagem sob a forma de um fragmento de história global que representa a confirmação de uma ou mais recepções, terminando em seguida.

7.3.4 Discussão

A correcção do hospedeiro e carregadores de difusão fiável é equivalente a que permitam traduzir e cumpram as restrições de fiabilidade de difusão e entrega descritas na Secção 5.5.2.

As restrições de difusão $\rho_{f,DE}$ são especificadas através dos parâmetros do carregador genérico, de acordo com:

$$a \in \textsc{Inc}(b) \Leftrightarrow id(a) \subseteq id(b)$$

e especificando no parâmetro *term* a recepção da mensagem por todos os destinatários. Estas restrições indicam quais as mensagens que têm que ser entregues exactamente uma vez.

Em primeiro lugar assegura-se que nenhuma mensagem é entregue mais do que uma vez. Por contradição, admite-se que uma mensagem a é entregue duas vezes por um processo. Então, durante a segunda entrega implica que no carregador de correspondente (Figura 7.4) a condição da linha 4 foi verdadeira e como tal que $f_g(id(a)) \not\subseteq H_g$. Como uma primeira entrega de a implica que $f_g(id(a))$ seja obrigatoriamente inserido em H_g (linhas 9 e 11), então conclui-se que a nunca terá sido entregue previamente, o que contradiz a hipótese.

Em segundo lugar, qualquer mensagem difundida por um processo correcto tem que ser entregue por qualquer um dos processos destinatários correctos que não entregue nenhuma outra que a substituia. Admitindo a difusão de uma mensagem a sem restrições de entrega por um processo correcto, então o mesmo processo recebe sempre o carregador difundido de volta, uma vez que qualquer canal c_{ii} é fiável.

Admita-se então como hipótese que um processo correcto destinatário da mensagem a nunca a entrega nem entrega qualquer outra que a substituia. Se um processo que nunca falha não entrega uma mensagem, então poderia ter sido porque:

1. a recebeu mas nunca a entregou.
2. nunca a recebeu;

O primeiro caso é impossível, directamente do algoritmo, pois em qualquer processo que nunca tenha entregue a mensagem implica que:

- ou não execute todos os passos o que é equivalente a ter falhado, o que contraria a hipótese;
- ou avalie como falsa a condição da linha 4, o que implica que entregou previamente uma mensagem b tal que $f_g(id(b)) \subseteq f_g(id(a))$ e $a \in \textsc{Inc}(b)$, ou seja, o processo entregou uma mensagem b que substitui a, o que contraria a hipótese;
- ou avalia como falsa a condição da linha 8, o que é equivalente ao processo em questão não ser destinatário da mensagem, o que contradiz a hipótese.

No segundo caso, como uma mensagem é sempre recebida pelo emissor que não falha, então ela é retransmitida um número ilimitado de vezes até que seja confirmada. Se é transmitida um número ilimitado de vezes então pelo propriedade de perda justa dos canais, ela, ou é inevitavelmente recebida, ou então o processo destino falha, o que contradiz a hipótese.

Uma restrição de entrega $\rho_{f,EE}$, indicando que uma mensagem que seja entregue a um dos seus destinatário, correcto ou não, seja inevitavelmente entregue a todos os restantes destinatários correctos, implica que a entrega seja atrasada até a recepção ser confirmada por uma maioria dos processos, uma vez que se assume que uma maioria de processos é correcta. Como consequência, estas restrições podem ser especificadas no parâmetro *ent* do carregador genérico.

Neste caso, a camada de entrega fiável apenas pode assegurar a entrega inevitável de uma mensagem se a aplicação garantir que a condição de entrega é inevitavelmente satisfeita, mesmo na presença de faltas. No entanto, é preciso que qualquer condição que seja cumprida por processos correctos seja reconhecida como tal por qualquer processo correcto.

Comece-se então por assegurar que a recepção de uma mensagem por um processo correcto implica a sua inevitável confirmação a qualquer outro processo correcto, o que serve também para garantir que uma mensagem bem recebida inevitavelmente deixe de ser retransmitida e armazenada. Para cada retransmissão existe uma probabilidade positiva de ser recebida e quando uma mensagem é recebida por um processo correcto uma confirmação é difundida. Como a confirmação tem também, pela propriedade de perda justa dos canais, uma probabilidade maior que zero de ser recebida, então existe uma probabilidade também positiva da confirmação ser recebida. Como podem ser feitas um número ilimitado de retransmissões, então a confirmação é invitávelmente recebida.

Admita-se então que uma condição de entrega é cumprida, ou seja, uma mensagem é recebida por uma maioria de processos processos correctos, mas que um processo correcto nunca conhece esse facto. Se isso acontece, é porque esse processo nunca recebeu a confirmação da recepção dessa mensagem por parte de um outro. Como a recepção de uma mensagem por um processo correcto implica a sua inevitável confirmação a qualquer outro processo correcto, chega-se à conclusão que:

- nenhum processo correcto recebeu a mensagem em causa e como tal a condição não é verdadeira, o que contraria a hipótese;
- não existe uma maioria de processos correctos, o que também contraria o modelo assumido.

Se uma mensagem é recebida por todos os seus destinatários e continua a ser retransmitida então é porque o processo que a retransmite não recebe uma confirmação. Como a recepção de uma mensagem implica a inevitável

recepção da confirmação correspondente por todos os processos correctos, então:

- ou a mensagem não foi recebida por algum dos destinatários, o que contraria a hipótese;
- algum dos destinatários não é correcto.

Para garantir que qualquer condição de terminação consistindo da recepção por um conjunto de processos é satisfeita, mesmo na presença de falhas de processos, é necessário um mecanismo de gestão de grupo que simule confirmações de recepção por parte de processos que falham.

7.4 Entrega ordenada

7.4.1 Algoritmos

A camada de entrega ordenada tem como objectivo impedir a entrega de pares de mensagens pela ordem errada, tal como especificado na Secção 5.5.3. Os algoritmos para entrega ordenada podem ser baseados no atraso ou então na aglomeração de mensagens [HT94], tanto na ordenação em relação à difusão bem como na ordenação em relação à entrega.

Os algoritmos baseados no atraso de mensagens armazenam, sem as entregar, quaisquer mensagens até que todas as suas predecessoras na ordem sejam recebidas e entregues. Quando isto acontece, qualquer mensagem armazenada é entregue e removida do armazenamento temporário.

No entanto, para assegurar que todas as mensagens recebidas são inevitavelmente entregues, um algoritmo deste tipo depende de comunicação fiável que assegure que todos os predecessores das mensagens recebidas são inevitavelmente entregues.

Os algoritmos baseados em aglomeração acrescentam a todas as mensagens enviadas todas as suas predecessoras na ordem, o que implica que na recepção de qualquer mensagem todas as suas predecessoras são também recebidas e em condições de ser entregues, se ainda não o foram.

Ao aglomerar as mensagens, um algoritmo de ordenação deste tipo torna-se ainda equivalente a um algoritmo de entrega fiável baseado em confirmação positiva, tornando impossível a concretização em separado destes dois aspectos. Outra semelhança com os algoritmos de confirmação positiva é a necessidade de receber confirmações para eliminar, das mensagens que são aglomeradas, aquelas que já foram recebidas.

Em algoritmos de ordenação em relação à entrega, em que é utilizado necessariamente um mecanismo sequenciador distribuído, como por exemplo um protocolo de consenso, a aglomeração significa que são as próprias mensagens o resultado produzido pelo sequenciador e não apenas os seus identificadores.

```
hospedeiro Ordenação exporta
  H_l,   /* fragmentos de história local (mensagens entregues) */
  seq    /* sequenciador distribuído */
fim
```

Figura 7.6: Componente hospedeiro para entrega ordenada.

O GROUPZ utiliza um mecanismo simples baseado no atraso de mensagens, com um sequenciador distribuído baseado num protocolo de consenso[3]. Como é baseado no atraso de mensagens depende de comunicação fiável para assegurar a entrega inevitável.

7.4.2 Hospedeiro

O hospedeiro de ordenação (Figura 7.6) exporta dois serviços para os carregadores:

- um conjunto H_l de fragmentos de história local descrevendo as entregas feitas localmente;
- um sequenciador distribuído *seq*, utilizado para transformar os fragmentos de história futura em fragmentos de história local.

O sequenciador garante que a ordenação de quaisquer duas mensagens é coerente com a ordem por que essas duas mensagens foram entregues ao serviço em pelo menos um dos processos.

O sequenciador é concretizado como um filtro para o consenso, mantendo para cada chave x de fragmentos de história futura (x, v), duas filas de espera:

- uma das sequências propostas;
- outras das sequências decididas.

Quando se tenta resolver um fragmento de história futura (x, v) para uma mensagem id então:

- (v, id) já existe na fila de espera decidida da chave x e então o resultado é a reunião de todos os id' tais que (v', id') precedem (v, id) na mesma fila e $v \vee v'$;
- (v, id) é inserida na fila de propostas correspondente a x e espera-se que apareça na fila de sequências decididas.

[3]Ver Apêndice C.

```
carregador Difusão para o hospedeiro Ordenação usa
      C,   /* carregador para a camada inferior */
      d,   /* dados transportados */
     id,   /* fragmento de história local (identificação) */
 dep_DE,   /* fragmento de história local (dependências da difusão) */
 dep_EE    /* fragmento de história futura (dependências da entrega) */
faz
   envia C(Entrega(d, id, dep_DE, dep_EE));
fim
```

Figura 7.7: Componente carregador para difusão ordenada.

```
carregador Entrega para o hospedeiro Ordenação usa
      d,   /* dados transportados */
     id,   /* fragmento de história local (identificação) */
 dep_DE,   /* fragmento de história local (dependências da difusão) */
 dep_EE    /* fragmento de história futura (dependências da entrega) */
faz
1    espera por dep_DE ∈ H_l;
2    x ← seq(id, dep_EE);
3    espera por x ⊆ H_l;
4    entrega d;
5    H_l ← H_l ∪ id;
6    H_l ← H_l ∪ x;
fim
```

Figura 7.8: Componente carregador genérico para entrega ordenada.

Uma actividade paralela propõe o conjunto das filas de espera propostas como valor para a iteração seguinte de consenso. Quando uma iteração é decidida, as filas decididas nessa iteração são acrescentadas às filas anteriormente decididas e os pares (id, v) nelas presentes removidos das filas de propostas.

A utilização do modelo de uma actividade por mensagem, em conjunto com o modelo adoptado para a camada de ordenação, em que uma mensagem espera por uma pré-condição, é entregue e em seguida avisa as pré-condições dela dependentes, permite que mensagens especificadas ao nível das condições de ordenação como concorrentes sejam entregues concorrentemente por actividades independentes.

7.4.3 Carregadores

O carregador de difusão ordenada (Figura 7.7) limita-se a enviar o carregador de entrega correspondente (Figura 7.8) que executa quatro operações:

- espera pela satisfação das dependências de ordem em relação à difusão especificadas como história local (linha 1);
- converte os fragmentos de história futura em história local (linha 2);
- espera pela satisfação dessas dependências (linha 3);
- entrega a mensagem (linha 4) e actualiza a história local (linhas 5 e 6).

Esta ordem de proceder é importante de modo a evitar o aparecimento de paradoxos de ordenação que poderiam surgir na situação em que duas mensagens devam ser ordenadas inequivocamente de acordo com dep_{DE}, mas também estejam relacionadas entre si, possivelmente por estarem ambas relacionadas por uma terceira, por dep_{EE}. Uma situação de bloqueio ocorre se a mensagem sucessora consegue resolver o seu fragmento de história futura antes da antecessora ter oportunidade de propor o seu, o que significa que a segunda mensagem é ordenada antes da primeira apesar de terem que ser ordenadas inversamente pela dependência de difusão.

Obrigando todas as mensagens a satisfazer as suas dependências de difusão antes de tentar resolver as de entrega, implica que em todos os pares relacionados pela difusão, a predecessora seja reconhecida como tal pelo sequenciador anulando a possibilidade de bloqueio.

7.4.4 Discussão

A correcção do hospedeiro e carregadores é equivalente a que permitam traduzir e cumpram as restrições de ordem descritas na Secção 5.5.3 e que quando composta com o protocolo de fiabilidade não perca ou duplique mensagens.

A especificação de restrições do formato $\rho_{o,DE}$ é feita especificando as dependência de cada mensagem, sabendo que:

$$a \in \textsc{Pred}(b) \Leftrightarrow id(a) \subseteq dep_{DE}(b) \vee (id(a) \subseteq dep_{DE}(c) \wedge c \in \textsc{Pred}(b))$$

De acordo com a definição de uma relação de causalidade, nunca se pode ter simultaneamente $a \in \textsc{Pred}(b)$ e $b \in \textsc{Pred}(a)$, o que é deixado a cargo da aplicação.

Considere-se então a entrega de qualquer par $a \in \textsc{Pred}(b)$, sem qualquer dependência de entrega, o que faz com que a actividade nunca espere nas linhas 2 e 3. Assuma-se que existe um processo que entrega b antes de a e de qualquer substituto de a.

Se isso aconteceu, então a actividade que fez a entrega de b previamente avaliou a condição da linha 1 como verdadeira, o que implica que:

- $id(a) \notin dep_{DE}(b)$ e portanto $a \notin \text{PRED}(b)$, contrariando a hipótese;
- $id(a) \in H_l$ e portanto uma actividade anteriormente inseriu $id(a)$ em H_l, o que significa que para acontecer implica que previamente tenha entregue a mensagem correspondente, mais uma vez contrariando uma hipótese.

Uma dependência de entrega $\rho_{o,EE}$ é representada por uma dependência de história futura:

$$\rho_{o,EE}(a,b) \Leftrightarrow \exists (x,v) \in dep_{EE}(a) : (x,v') \in dep_{EE}(b) \wedge (v \vee v')$$

o que dá origem a dois casos distintos:

1. duas mensagens estão apenas relacionadas por dependências de entrega, não podendo ser entregue por ordem inversa em processos distintos;
2. duas mensagens estão relacionadas também por uma dependência de difusão, não podendo ser entregues por ordem inversa à especificada nessa dependência.

No primeiro caso, sendo recebida uma das duas mensagens de um par relacionado uma dependência de entrega, então durante a resolução através do sequenciador:

- a mensagem em questão já está numa das filas decididas, pelo que pode imediatamente ser convertida numa dependência de difusão, pois:
  - a outra mensagem ainda não aparece nas filas decididas e como tal, ou já foi decidida e entregue, já não sendo possível violar a restrição, ou ao ser proposta depois da decisão desta não pode aparecer na sua história passada;
  - aparecem ambas nas filas e portanto podem imediatamente ser calculadas as dependências de ambas.
- a mensagem não está numa fila decidida, pelo que é proposta e espera-se pela decisão, que inevitavelmente acontece ficando reduzido ao primeiro caso.

No segundo caso, em que $a \in \text{PRED}(b)$ como as dependências de difusão são satisfeitas antes de as dependências de ordem serem propostas ao serviço sequenciador, então, a mensagem b em todos os processos é proposta ao sequenciador depois de a ter sido entregue, pelo que a ordem decidida pelo sequenciador é sempre coerênte com a dependência de difusão.

A condição de não duplicar mensagens é trivialmente satisfeita, uma vez que cada mensagem está associada a uma actividade, cada actividade só a entrega no máximo uma vez e não são criadas novas actividades. A condição de não perder mensagens pode acontecer por:

1. terminação da actividade associada à mensagem sem que esta seja entregue, o que dado o algoritmo é trivialmente impossível sem que ocorra uma falha do processo;

2. bloqueio eterno de uma actividade.

A segunda situação não pode ser evitada, caso seja recebida uma mensagem cujas dependências não sejam satisfeitas. Para evitar esta situação é necessário que:

- a aplicação garanta que na ausência de falhas de processos, todas as dependências sejam inevitavelmente satisfeitas;

- no caso da falha de um processo, que pode levar à perda de uma mensagem essencial para satisfazer dependências de ordem, essa falta seja reparada.

A reparação deste tipo de faltas pode ser efectuada por intermédio de um sistema de gestão da composição de um grupo que satisfaça as dependências de ordem, como o que é apresentado na secção seguinte.

7.5 Gestão do grupo

7.5.1 Algoritmos

Um mecanismo de gestão de grupo tem como função assegurar a vivacidade do sistema pela exclusão do grupo de processos que falham e pela inclusão de processos correctos. Para que a modificação da composição do grupo seja vista de forma coerente por todos os membros do grupo, têm que ser respeitadas as restrições quanto à vista em que cada mensagem pode ser entregue, tal como especificado na Secção 5.5.4.

Para o efeito, um mecanismo de gestão de grupo tem que assegurar [Mal95]:

- acordo quanto aos elementos que compõem o grupo;

- acordo quanto à vista do grupo em que cada mensagem é recebida.

Para o primeiro, pode ser utilizado um protocolo de consenso ou outro mecanismo equivalente, como a ordenação total.

O segundo é assegurado por um protocolo de esvaziamento, que consiste em assegurar que todas as mensagens incompatíveis com a vista futura são

entregues em todos os membros do grupo antes de se efectuar a mudança. Para o efeito, as tentativas de enviar novas mensagens enquanto se processa uma mudança de vista são bloqueadas. Deste modo, por meio de um mecanismo de confirmação positiva, qualquer processo que pretenda instalar uma nova vista tem a certeza que todos os restantes elementos dessa vista receberam durante a vista anterior todas as mensagens que vão ser incompatíveis com a vista futura. Quando o protocolo de esvaziamento está concluído, a nova vista é proposta e inevitavelmente decidida através do protocolo de consenso.

Há ainda a considerar o mecanismo utilizado para despoletar mudanças de vista, tanto para entrada como para expulsão de processos, uma vez que dele depende a correcção das condições de vivacidade do sistema. Para este efeito pode ser utilizado um detector de falhas imperfeito[4] [SR93], o que tem no entanto a limitação de não relacionar a expulsão de um membro com a semântica da aplicação. Por exemplo, no caso de um trinco distribuído apenas faz sentido expulsar membros que detenham o trinco, o que não é possível com um mecanismo deste tipo.

O GROUPZ utiliza um protocolo de esvaziamento em conjunto com ordenação total para suportar grupos não particionáveis. No entanto, dadas a possibilidade de configurar o comportamento das mensagens em relação às vistas, o protocolo de esvaziamento não é total. Além disso, o critério de iniciação de mudanças de vista é parametrizável pela aplicação.

Uma vez que a gestão da vista é no GROUPZ dependente apenas do funcionamento normal das camadas de ordenação e fiabilidade, não sendo necessárias operações especiais para levar a cabo o protocolo de esvaziamento, é ainda um exemplo da utilidade de protocolos configuráveis. Neste caso, da configurabilidade das camadas de ordenação e fiabilidade.

7.5.2 Hospedeiro

Uma vista é formada por um conjunto de membros efectivos e dois outros conjuntos correspondentes aos membros a expulsar e aos não membros a incluir. A estes dois conjuntos chama-se a incerteza sobre a vista. As vistas com incerteza são por natureza transitórias e na ausência de pedidos para entrar e sair do grupo, a vista estabiliza numa situação sem incerteza.

Qualquer processo pode aumentar em qualquer altura os conjuntos relativos à incerteza. No entanto, estes conjuntos apenas podem ser diminuídos pelas mudanças de vista, que são totalmente ordenadas entre si.

Qualquer mensagem ao ser enviada, é etiquetada com a vista do processo na altura em que é emitida. Deste modo podem avaliar-se as dependências de difusão, que são armazenadas no processo emissor. Ao ser recebida, uma mensagem armazena no receptor as dependências de entrega.

[4] Ver Secção 2.5.2 e [CT94].

```
hospedeiro Vista exporta
        p,   /* nome do processo local */
        V,   /* conjunto de processos que formam a vista */
       V+,   /* incerteza quanto ao crescimento do grupo */
       V-,   /* incerteza quanto ao encolhimento do grupo */
       Dd,   /* dependências de difusão armazenadas */
       De,   /* dependências de entrega armazenadas */
     ΔV+,    /* conjunto de processos a acrescentar à vista */
     ΔV-,    /* conjunto de processos a retirar da vista */
       md,   /* conjunto de dependências de difusão */
       me,   /* conjunto de dependências de entrega */
     conf,   /* conjunto de processos que confirmaram a mudança */
  mudando    /* mudança de vista em curso */
fim
```

Figura 7.9: Componente hospedeiro para entrega coerente com a vista do grupo.

O protocolo de esvaziamento usa as dependências armazenadas no emissor e no receptor para calcular quais as mensagens que não podem ser entregues na vista futura e como tal têm que preceder a mudança da vista. Ou seja, as restrições em relação à vista do grupo transformadas em dependências de ordem em relação às mudanças de vista, que são cumpridas pelas camadas inferiores.

Como consequência, o protocolo de esvaziamento do grupo não é total uma vez que admite que:

- mensagens possam ser difundidas durante o período de incerteza, desde não sejam incompatíveis com nenhuma das vistas que possam resultar dessa incerteza;
- algumas mensagens não sejam serializadas com a instalação da nova vista, podendo ser entregues por diferentes processos em vistas distintas.

As mudanças de vista são iniciadas em resposta a qualquer situação em que a manutenção da vista actual pode levar ao bloqueio de algum processo. Esta situação é detectada por carregadores ao nível dos vários hospedeiros, podendo esses carregadores ser genéricos ou especialmente adaptados a aplicações concretas.

Quando algum processo provoca alguma destas situações e é suspeito, dá-se início a uma mudança de vista que o excluia. Deste modo, apenas

os processos efectivamente responsáveis pelo bloqueamento do sistema são expulsos, sendo sempre conhecida a razão concreta da sua expulsão.

Os serviços disponibilizados pelo hospedeiro de vistas (Figura 7.9) são utilizados por em dois tipos de carregadores. Nomeadamente, para os carregadores de mensagens, que especificam as restrições em termos de vista para entrega de mensagens e para os carregadores de vistas, que alteram a vista actual. Estes serviços são:

- o identificador do processo local;
- um conjunto de identificadores de processos V, representando a vista do grupo por parte do processo;
- dois conjuntos de identificadores de processos V^- e V^+, representando a incerteza quanto à vista actual, respectivamente, quanto aos processos que podem vir a ser expulsos e quanto aos processos que podem vir a ser incluídos;
- conjuntos de dependências de difusão das mensagens difundidas localmente D_d de entrega das mensagens entregues localmente D_e;
- conjunto de processos ΔV^- e ΔV^+ que por iniciativa deste processo estão a ser excluídos ou incluídos na vista;
- conjuntos de dependências m_d e m_e, reunidas durante o processo de mudança de vista;
- um conjunto de identificadores de processos $conf$ que confirmaram a mudança pedida enviando as suas dependências;
- um valor lógico *mudando* que assinala uma tentativa de mudança de vista por parte do processo local.

7.5.3 Carregadores

Cada dependência de vista é concretizada como uma função de três conjuntos de identificadores de processos. Estes parâmetros representam a composição da vista actual e a incerteza sobre uma possível vista futura. O resultado é um conjunto de dependências de ordem a serem impostas à mudança de vista para cumprir a restrição correspondente.

O carregador para difusão de uma mensagem (Figura 7.10) cumpre as seguintes tarefas:

- averigua se as restrições de vista da mensagem podem ser avaliadas correctamente na actual situação de incerteza quanto à vista, tendo no pior caso que esperar por uma situação sem incerteza (linha 1);

```
carregador Difusão para o hospedeiro Vista usa
      C,   /* carregador para a camada inferior */
      d,   /* dados transportados */
  dep_DE,  /* dependências da vista de difusão a acrescentar */
  dep_EE,  /* dependências da vista de entrega a acrescentar */
    n_DE,  /* dependências da vista de difusão a eliminar */
    n_EE,  /* dependências da vista de entrega a eliminar */
     id,   /* identificação da mensagem */
      N    /* carregador para notificação da vista de difusão */
faz
1     espera por ∀dep ∈ dep_DE, dep(V, V+, V−) = ∅;
2     D_d ← (D_d ∪ dep_DE) \ n_DE;
3     entrega N(id, V);
4     envia C(Entrega(d, V, dep_DE, dep_EE, n_DD));
fim
```

Figura 7.10: Componente carregador para envio para grupo.

- armazena no hospedeiro as suas dependências de difusão ao mesmo tempo que remove outras que tenham ficado obsoletas (linha 2);
- notifica a aplicação da vista em que a mensagem foi difundida, que determina em parte qual a vista em que será entregue (linha 3);
- envia a mensagem para as camadas inferiores (linha 4).

O carregador de entrega (Figura 7.11), quando recebido no hospedeiro de vista em cada um dos destinatários, efectua as seguintes tarefas:

- espera por uma situação em que a incerteza não afecte esta mensagem (linha 1);
- confirma que as condições relativas à difusão são válidas (linha 2), uma vez que no caso de incorrecção do processo emissor pode chegar aqui uma mensagem numa vista incompatível, caso em que não é entregue;
- caso contrário, actualiza as dependências de entrega (linha 3) e entrega a mensagem (linha 4).

Como durante a entrega se verifica se a incerteza afecta a mensagem, pode acontecer que:

- não afecte, e a mensagem possa ser entregue em qualquer das vistas resultantes da incerteza actual;

```
carregador Entrega para o hospedeiro Vista usa
      d,   /* dados transportados */
     V_0,  /* vista de difusão */
  dep_DE,  /* dependências da vista de difusão */
  dep_EE,  /* dependências da vista de entrega */
    n_EE   /* dependências da vista de entrega */
faz
1     espera por ∀dep ∈ dep_EE, dep(V, V+, V−) = ∅;
2     se ∀dep ∈ dep_DE, dep(V_0, (V ∪ V+) \ V_0), (V \ V_0) ∩ V− = ∅ então
3        D_e ← (D_e ∪ dep_EE) \ n_EE;
4        entrega d;
5     fim
fim
```

Figura 7.11: Componente carregador para entrega a um grupo.

```
carregador Inicial para o hospedeiro Vista usa
  V_0   /* conjunto de processos (vista inicial) */
faz
1     V ← V_0;
2     V+ ← ∅;
3     V− ← ∅;
fim
```

Figura 7.12: Componente carregador para inicialização de um grupo.

- afecta a mensagem, o que dá origem a duas situações:
  - pelo menos um dos processos correctos entrega a mensagem antes de concluir o protocolo de recolha de dependências, obrigando a que a dependência seja respeitada por todos os processos, que como tal é entregue por todos na vista mais antiga;
  - seja bloqueada até que seja instalada uma vista sem incerteza, que necessariamente não contém essa mensagem nos seus predecessores.

Entre os carregadores para gestão da vista, o mais simples é o que instala incondicionalmente uma nova vista (Figura 7.12), que pode ser usado quer para criar um grupo apenas com um elemento, ao qual outros processos se juntam, quer para inicializar estaticamente um grupo.

Em qualquer outra situação, a mudança de vista pressupõe o esvaziamento do grupo. Para o efeito, um processo que pretende propor uma nova vista envia a todos os processos que fazem parte tanto da vista actual como

```
carregador PedirDependências para o hospedeiro Vista usa
  V_m^+,   /* conjunto de processos (a acrescentar à vista) */
  V_m^-    /* conjunto de processos (a retirar da vista) */
faz
1     V^+ ← V^+ ∪ V_m^+;
2     V^- ← V^- ∪ V_m^+;
3     envia DevolverDependências;
p, V^+, V^-, D_d, D_e fim
```

Figura 7.13: Componente carregador para iniciar o esvaziamento.

```
carregador DevolverDependências para o hospedeiro Vista usa
    j,   /* identificador de processo */
  V_m^+, /* conjunto de processos (a acrescentar à vista) */
  V_m^-, /* conjunto de processos (a retirar da vista) */
   d_d,  /* dependências de difusão */
   d_e   /* dependências de entrega */
faz
1     se ΔV^+ ⊆ V_m^+ ∧ ΔV^- ⊆ V_m^- então
2        conf ← conf ∪ {j};
3        m_d ← m_d ∪ d_d;
4        m_e ← m_e ∪ d_e;
5     fim
fim
```

Figura 7.14: Componente carregador para completar o esvaziamento.

da vista futura um carregador (Figura 7.13) que quando recebido na mesma vista:

- informa o receptor da situação de incerteza quanto à vista (linhas 1 e 2);
- recolhe do receptor informação quanto às dependências da vista que está prestes a ser terminada (linha 3).

O processo de esvaziamento completa-se com o recebimento por parte do processo iniciador do carregador com a informação sobre as dependências relevantes (Figura 7.14). Esta informação consiste num conjunto de dependências de vista que quando recebidas pelo processo proponente são acrescentadas ao conjunto de dependências que terão que ser respeitadas durante a mudança (linhas 3 e 4), ou seja, efectivamente transformadas em dependências de ordem. Esta operação, é registada no conjunto de processos dos quais foi recebida a confirmação (linha 2).

```
carregador IniciarMudança para o hospedeiro Vista usa
  V_m^+,   /* conjunto de processos (a acrescentar à vista) */
  V_m^-    /* conjunto de processos (a retirar da vista) */
faz
1     se V_m^+ ⊈ V ∨ V_m^- ⊆ V então
2        ΔV^+ ← (V_m^+ \ V) ∪ ΔV^+;
3        ΔV^- ← (V_m^- ∩ V) ∪ ΔV^-;
4        conf ← ∅;
5        envia PedirDependencias(id, ΔV^+, ΔV^-);
6        se ¬mudando então
7           mudando ← V;
8           inicia EfectuarMudança;
9        fim
10    fim
fim
```

Figura 7.15: Componente carregador para iniciar a mudança de vista.

Quando uma situação em que é necessário mudar a vista do grupo é detectada num processo, um carregador encarregado de iniciar essa mudança (Figura 7.15) é executado na camada de gestão de vistas. Depois de verificar que a operação que pretende executar é válida (linha 1), esse carregador:

- acrescenta os identificadores dos processos a excluir ou incluir na vista ao estado do hospedeiro (linhas 2 e 3), limpando o conjunto de processos dos quais foi recebida confirmação (linha 4);
- comunica a todos os outros processos os limites de incerteza quanto à vista actual, pedindo ao mesmo tempo as dependências relevantes (linha 5);
- se o processo local não está já a efectuar uma mudança de vista, iniciar o carregador correspondente (linhas 6 a 9).

O carregador que efectua a mudança (Figura 7.16) instala-se no hospedeiro onde foi iniciado e espera que todos os processos que vão mudar de vista respondam com as dependências relevantes para essa situação de incerteza ou então que qualquer outra mudança de vista tenha sucesso (linha 1).

No caso de entretanto não se ter efectuado qualquer outra mudança de vista que torne o processo desnecessário (linha 2), então:

- transforma as dependências de vista em termos de difusão em dependências de ordem também de difusão (linha 3);

```
carregador EfectuarMudança para o hospedeiro Vista faz
1     espera por (∀p ∈ V \ ΔV⁻ : p ∈ conf) ∨ (ΔV⁺ = ∅ ∧ ΔV⁻ = ∅);
2     se ¬(ΔV⁺ = ∅ ∧ ΔV⁻ = ∅) então
3        dep_d ← {x : dep ∈ m_DE ∧ x = dep(V, ΔV⁺, ΔV⁻)};
4        dep_e ← {x : dep ∈ m_EE ∧ x = dep(V, ΔV⁺, ΔV⁻)};
5        n_DE ← {dep : dep ∈ m_DE ∧ dep(V, ΔV⁺, ΔV⁻) ≠ ∅};
6        n_EE ← {(Vista, V)}∧{dep : dep ∈ m_EE∧dep(V, ΔV⁺, ΔV⁻) ≠ ∅};
7        envia C(Mudança(ΔV⁺, ΔV⁻, n_DE, n_EE));
8     fim
9     mudando ← F;
fim
```

Figura 7.16: Componente carregador para mudança de vista.

- transforma as dependências de vista em termos de entrega em dependências de ordem também de entrega, às quais acrescenta a dependência de entrega em relação à vista corrente (linha 6);
- notifica todos os restantes processos quanto à nova vista (linha 7).

Esta notificação, é feita com um carregador de difusão ordenada que é parametrizado com dep_d e dep_e. Como consequência, inclui pelo menos uma dependência de ordem de entrega $(Vista, \mathcal{V})$, o que obriga à resolução do consenso com todos os membros da vista actual para instalar a nova vista. Este carregador de mudança implica ainda um carregador de difusão fiável parametrizado com:

- um identificador único;
- uma confirmação forjada da recepção de todas as mensagens usadas para efectuar o esvaziamento;
- uma confirmação forjada de todas as mensagens enviadas pelos processos que estão a ser expulsos;
- a entrega a uma maioria da vista anterior como condição de entrega;
- a entrega à totalidade da vista futura como condição de terminação.

Um carregador de mudança de vista (Figura 7.17) resolve uma situação de incerteza quanto a uma vista modificando o conjunto de identificadores de processos de acordo com um conjunto de processos a acrescentar e outro conjunto de processos a remover.

Ao mesmo tempo, este carregador remove em cada um dos processos as dependências que se tornaram obsoletas com esta mudança de vista. De

```
carregador Mudança para o hospedeiro Vista usa
 ΔV+,   /* conjunto de processos a acrescentar à vista */
 ΔV−,   /* conjunto de processos a retirar da vista */
  n_DE, /* conjunto de dependências de difusão */
  n_EE  /* conjunto de dependências de entrega */
faz
1    V ← (V ∩ V_m^−) ∪ ΔV+_m;
2    D_d ← D_d \ n_DE;
3    D_e ← D_e \ n_EE;
4    V+ ← V+ \ V;
5    V− ← V− ∩ V;
6    ΔV+ ← ΔV+ \ V;
7    ΔV− ← ΔV− ∩ V;
fim
```

Figura 7.17: Componente carregador para mudança de vista.

```
carregador Notificação para o hospedeiro * usa
 N,   /* carregador para notificação */
  c,  /* condição de notificação */
  D   /* detector de falhas imperfeito */
faz
1    espera por c(p) ∧ p ∈ D;
2    entrega N(p);
fim
```

Figura 7.18: Componente carregador para notificação de situação de mudança de vista.

modo a assegurar que este conjunto não cresce para sempre é necessário que inevitavelmente:

- ou exista uma mudança de vista que seja incompatível com qualquer dependência;
- ou seja difundida uma mensagem sucessora na ordenação de modo que a suas dependências substituiam, por transitividade, as da antecessora.

Uma mudança de vista para inclusão de um novo processo é despoletada pela chegada a um qualquer processo membro da vista de uma mensagem do processo que pretende entrar. Essa mensagem é constituída por um carregador que inicia a mudança de vista com o conjunto contendo o processo em questão como parâmetro.

A expulsão de um processo pode ser despoletada pela aplicação de uma forma semelhante. Em alternativa, pode resultar da detecção de uma condição de bloqueio em qualquer das camadas, detectada por um carregador de diagnóstico (Figura 7.18). Por exemplo:

- a falta de confirmação da recepção de uma mensagem por parte de um processo faz com que essa mensagem tenha que ser armazenada para sempre;
- a falta de recepção de uma mensagem que faz parte das dependências de ordem de um outra impede que seja entregue.
- um dos processos que deve passar à próxima vista, ao não comunicar aos restantes as dependências de vista, não consegue concluir o protocolo de esvaziamento.

7.5.4 Discussão

A correcção do funcionamento da camada de gestão de vistas significa que:

- não viola as garantias de ordem e fiabilidade asseguradas pelas camadas inferiores;
- permite especificar e faz cumprir as restrições de entrega em termos de vista do grupo tal como especificadas na Secção 5.5.4;
- permite assegurar as condições de vivacidade nas camadas de ordem e fiabilidade, garantindo que pedidos de mudança de vista são inevitavelmente efectuados.

Começando pelas condições de não trivialidade, quando um carregador é recebido num processo correcto pelo hospedeiro de vista, com uma actividade associada, então pode acontecer que:

1. as suas condições de difusão são compatíveis com a vista actual e então, pelo algoritmo, a mensagem é inevitavelmente entregue, uma vez que não há hipótese da actividade correspondente ser bloqueada ou terminada, a menos de falha que se assume não existir;
2. as suas condições de difusão não são compatíveis com a vista actual e mensagem não é entregue.

Se no primeiro caso não há dúvidas quanto à correcção, no segundo caso é necessário garantir uma das seguintes condições é verdade para não contrariar o aspecto de fiabilidade do protocolo:

- foi entregue outra mensagem que a substitui;

- a mensagem foi enviada por um processo incorrecto.

Se o processo emissor é considerado incorrecto, então não há de facto qualquer obrigação de entregar a mensagem. Se o processo é considerado correcto então pertence tanto à vista actual como à anterior e como tal as suas dependências de difusão foram incluídas no processo de mudança de vista e como tal:

- incluiam a mensagem em questão, pelo que a vista é atrasada e as dependências de difusão da mensagem serão obrigatoriamente satisfeitas durante a entrega;
- não incluiam a mensagem em questão, o que significa que nesse processo foi entregue uma outra mensagem que substituia essa tornando então possível a sua não entrega.

Neste último caso, é da responsabilidade da aplicação assegurar que a nova mensagem:

- é enviada pelo mesmo processo e especifica dependências de difusão concordantes com a anterior;
- é enviada por outro processo e especifica dependências de entrega concordantes com a anterior.

O respeito pela ordenação das mensagens está garantido pelo facto de, directamente do algoritmo, ser óbvio que as mensagens são entregues pela mesma actividade em que foram recebidas, pelo que, a ordenação de duas recepções corresponderá à consequente ordenação de duas entregas, sem prejuízo da concorrência possível entre mensagens não ordenadas.

Quanto à correcção de uma entrega numa vista compatível, é preciso considerar a correcção em relação à difusão e em relação a outras entregas. A impossibilidade de uma mensagem ser entregue numa vista incompatível com aquela em que foi difundida violando uma restrição do tipo ρ_{DE} é trivialmente assegurada pela condição da linha 1 do carregador de entrega (Figura 7.11).

Assuma-se então uma mensagem é entregue numa vista incompatível com a vista em que foi entregue em outro processo. Se isso aconteceu então foi porque entre essas duas entregas[5] existiu uma ou mais mudanças de vistas em que pelo menos uma muda para uma vista incompatível. Tome-se então a primeira dessas mudanças.

Se existe uma mensagem que em processos diferentes é entregue antes e depois da mudança de vista então essa mudança de vista não inclui nas suas dependências de ordenação essa mensagem. Isso pode acontecer porque:

[5]Dada a ordenação total das vistas, quaisquer duas entregas em vistas diferentes estão também totalmente ordenadas entre si.

- ou nenhum dos processos constantes da vista futura tinha tinha armazenado a dependência;
- ou tinha armazenado a dependência mas não a comunicou durante a mudança.

O segundo caso é obviamente impossível pois cada processo ou comunica todas as dependências relevantes ou não comunica nenhuma, e o processo de mudança de vista só é possível quando são recebidas as dependências de todos os processos constantes da vista futura.

O primeiro caso pode acontecer em duas situações:

- nenhum dos processos da vista futura tinha entregue previamente a mensagem, portanto, ela só podia ter sido entregue por processos expulsos, o que contraria a hipótese;
- algum processo da vista futura já a tinha entregue mas tinha já removido a dependência relevante.

Se um processo tinha já removido a dependência relevante, isso só pode ter acontecido porque:

- foi entregue outra mensagem que a removeu;
- ela foi removida por uma mudança de vista prévia.

Ora se existiu uma mudança de vista prévia nessas condições é porque estava ordenada com essa mensagem, o que significa ainda que foi uma mudança para uma vista incompatível com a entrega inicial o que contraria a hipótese de a mudança considerada ser a primeira mudança para uma vista incompatível com a entrega inicial.

Se foi entregue outra mensagem que a removeu, então é porque essa mensagem é sucessora na ordenação da mensagem em causa e tem as mesmas dependências em termos de vista, pelo que essas dependências são obrigatoriamente incluídas durante a mudança em causa.

A vivacidade das vistas significa que a partir do momento em que é pedida a expulsão de um processo por um outro pertencente a uma maioria de processos correctos, ele é inevitavelmente expulso [6].

Se é feito um pedido de mudança de vista por um processo correcto então é porque o carregador que inicia a mudança de vista é executado num processo correcto. Assuma-se então que uma vista que satisfaça os requisitos desse pedido não é instalada. Isso pode acontecer porque na Figura 7.17:

[6]Este pressuposto foi tomado para simplificar esta primeira concretização e deve no futuro ser relaxado para inevitavelmente expulsar um processo apenas se a partir de algum momento existe sempre pelo menos um processo correcto que pede essa expulsão.

- a condição da linha 1 não é satisfeita, o que implica que a vista actual cumpre os requisitos do pedido, contrariando a hipótese;
- o carregador da mudança enviado na linha 7 da Figura 7.16 não é recebido, o que pelas propriedades de ordenação e fiabilidade só é possível se não existe uma maioria de processos correctos, o que contraria a hipótese.

Ou então o carregador se bloqueia para sempre na condição da linha 1 da Figura 7.16. Se é este o caso é porque existe um processo pertencente a $V \setminus V^-$ do qual nunca nunca é recebida a confirmação. Uma vez que tanto o pedido com a resposta com as dependências são enviadas num canal fiável isso significa que:

- o processo em questão falhou;
- foi entretanto instalada outra vista.

Se ele falhou, então é inevitavelmente suspeito e como existe pelo menos uma actividade bloqueada devido a ele, então é proposta uma nova vista que não o inclui. Se a sua expulsão é proposta, então ele é incluído no conjunto V^- e como tal a sua falha deixa de ser razão para que a actividade se bloqueie para sempre na linha 3.

Se a resposta não foi recebida porque entretanto foi mudada a vista, então é porque essa vista satisfaz os requisitos da mudança em curso, pelo que pode ser cancelada.

7.6 Análise de desempenho

Ao contrário da execução de uma linguagem como o C++ [Str91], em que o desempenho do código é apenas afectado pelo desempenho do processador e em menor escala pela optimização conseguida pelo compilador, o desempenho de um programa em JAVA é fortemente influenciado pelo tipo de máquina virtual onde se executa, uma vez que é utilizada um linguagem intermédia.

Existem três categorias principais de máquina virtual para tradução e execução de código intermédio:

- interpretador de código intermédio;
- compilador dinâmico de código intermédio, que efectua a tradução de código intermédio apenas uma vez para cada segmento de código e apenas quando necessário durante a execução, não podendo efectuar qualquer optimização global ou demorada;

	Interpretador	*Compilador dinâmico*	*Pré-compilador*
Entrada/Saída	1	1.5	1.5
Processamento	1	19	55
Média	1	10	50

Tabela 7.1: Comparação do desempenho de diversas máquinas virtuais. Números maiores significam velocidades superiores.

	2	*3*	*4*
Não fiável	10	10	11
Fiável	83	105	130
Ordenada (DE)	98	118	145
Ordenada (EE)	462	790	1285
Vista	478	823	1346

Tabela 7.2: Variação dos tempos de ida e volta de uma mensagem com grupos de 2 a 4 processos.

- pré-compilador de código intermédio, que processa todo o código intermédio antes da execução, permitindo efectuar uma optimização global e morosa do código gerado.

A Tabela 7.1 apresenta uma comparação da velocidade medida de diferentes máquinas virtuais para aplicações típicas, como cópia de ficheiros, um compilador e um jogo computacionalmente intensivo. Em termos médios, pode assumir-se que um compilador dinâmico é em média dez vezes mais rápido que o interpretador de referência e um pré-compilador optimizante em média cinquenta vezes mais rápido [MMBC97]. Outros autores defendem mesmo que um compilador dinâmico optimizado poderá ser até sete vezes mais rápido que um compilador dinâmico típico e praticamente tão eficiente quanto uma linguagem como o C++ [KG97].

A escolha da linguagem JAVA implica então uma penalização significativa em termos de desempenho nas plataformas onde apenas o interpretador de referência se encontra disponível.

A Tabela 7.2 apresenta então um conjunto de tempos de transmissão de mensagens entre os dois membros de um grupo. As medidas de desempenho apresentadas foram obtidas com o interpretador de referência, devido ao facto de não se encontrar disponível nenhum pré-compilador ou compilador dinâmico suportando todas as características necessárias, como múltiplas actividades e difusão selectiva, nas plataformas disponíveis para testes.

Para a medição dos tempos de ida e volta de mensagens foi utilizado um multiprocessador (4× PENTIUM PRO 200) com LINUX 2.0, JDK 1.1.3 (interpretador de referência) e difusão simulada. Deste modo, as medições de tempos com grupos até quatro processos aproximam o que aconteceria com quatro máquinas independentes. De uma análise dos resultados obtidos pode observar-se que:

- o tempo de ida e volta de uma mensagem correspondendo a uma estrutura de objectos extremamente simples é significativamente superior aos valores normais para a passagem de um datagrama em situações semelhantes, por uma ordem de magnitude;
- são especialmente significativos os tempos correspondentes à passagem para comunicação fiável e para ordem total;
- os tempos relativos à ordenação total crescem rápidamente com o número de elementos do grupo.

Convém então procurar explicar estas observações de modo a poder obter melhores resultados.

A utilização de camadas de serialização dedicadas a aplicações e protocolos concretos, já referida a propósito da concretização de protocolos cuja representação externa está normalizada, é a optimização mais interessante em termos de desempenho, como se conclui do tempo necessário para passagem não fiável de uma mensagem.

O tempo obtido numa transmissão fiável de uma mensagem pode ser também em grande parte atribuído ao processo de serialização, pois não só são trocadas várias mensagens até que os dados possam ser entregues como também essas mensagens são estruturas mais complexas e de serialização mais demorada.

Uma outra possibilidade de optimização dos protocolos relacionada com a serialização, independente do facto desta ser genérica ou dedicada, é o armazenamento para possível reutilização das versões serializadas dos objectos que constituem as mensagens. Esta optimização pode ser bastante útil quando há necessidade de retransmitir mensagens por motivos de perdas mas sobretudo na transmissão da pulsação dos processos, onde a tarefa de serialização da pulsação é sistematicamente repetida.

No modelo utilizado, que associa uma actividade a cada mensagem, é ainda relevante a sobrecarga resultante das operações de sincronização no acesso pelos componentes carregadores aos serviços dos hospedeiros. Neste contexto, será possivelmente vantajosa em termos de desempenho a substituição da sincronização implícita da linguagem JAVA por primitivas invocadas explicitamente que não resultem num tão grande número de operações de sincronização.

Embora a minimização de cópias seja um dos aspectos mais importantes na optimização de concretizações tradicionais de protocolos, essa preocupação não faz sentido em protocolos por objectos, a não ser aos níveis mais baixos. Isto acontece porque sendo a mensagem em trânsito um grafo de objectos ligados por referências, apenas as referências são modificadas não havendo nunca necessidade de fazer cópias dos dados.

Por outro lado, passa a ser relevante a sobrecarga resultante da criação e destruição de objectos, relacionada sobretudo com:

- a reserva e libertação de espaço de memória
- a invocação de construtores aninhados.

Na linguagem JAVA, ambos estes aspectos estão fora do controlo do programador, pelo que as possibilidades de optimização são restritas.

A utilização de uma linguagem como o C++ permitiria um controlo mais apertado destes aspectos, por exemplo, pela utilização de um gestor de memória dedicado e da supressão das chamadas a construtores.

Uma questão importante para o desempenho do protocolo de ordenação na presença de restrições de entrega, que é traduzido pelos números apresentados, é a sobrecarga introduzida pelo protocolo de consenso. Esta situação poderá ser aliviada com recurso a:

- optimização do tempo de ida e volta de mensagens fiáveis;
- um protocolo de consenso que não necessite de comunicação fiável[7];
- um serviço de consenso presente em apenas alguns dos processos e utilizado pelos restantes [GS96a].

Numa situação com maior número de processos, poderia ainda ser preocupante a latência correspondente à rotação do coordenador no protocolo de consenso. Embora esta esteja atenuada pelo facto de as propostas serem feitas durante a entrega das mensagens e como tal, por qualquer processo, será desejável a concretização de um algoritmo, tal como o *acordo uniforme sobre prefixos* [Anc96], que não apresenta esta limitação.

Nesta análise de desempenho convém ainda referir que a comparação com um protocolo não configurável de difusão não é totalmente correcta pois um protocolo configurável é, em testes sintéticos como estes, injustamente penalizado porque:

- não se entra em conta com o facto que o protocolo configurável apresenta um valor acrescentado em termos de funcionalidade que num protocolo não configurável tem forçosamente que ser obtida ao nível da aplicação, não sendo portanto medida nestes testes;

[7] Ver Secção 2.5.2.

- os protocolos em questão intencionalmente são prejudicados no desempenho para reduzir a possibilidade de bloqueio, o que num sistema em grande escala penalizaria o protocolo não configurável.

Uma análise objectiva do desempenho de um protocolo configurável teria pois que passar pela avaliação de aplicações concretas num sistema em grande escala. Na indisponibilidade de recursos para o fazer, seria aconselhável proceder à simulação de um tal sistema que permitisse análises exaustivas de aplicações completas.

7.7 Conclusão

Neste capítulo apresentou-se a concretização de um protocolo configurável de difusão, tal como especificado no Capítulo 5. Sendo esta concretização baseada nas infra-estruturas para protocolos abertos orientados por objectos descritas no Capítulo 6, consiste então na definição de uma rede de difusão de objectos, num conjunto de componentes hospedeiros para cada um dos aspectos do protocolo e finalmente de um conjunto de componentes carregadores para cada um desses aspectos.

Para cada um aspectos do protocolo são descritos os algoritmos concretizados, a sua partição em componentes hospedeiro e carregadores, sendo então explicada a sua adequação à especificação.

Na concretização dos diferentes aspectos do protocolo, como se pode encarar a camada de gestão do grupo como uma aplicação das camadas subjacentes, observaram-se as vantagens da concretização aberta orientada por objectos de protocolos ao nível da funcionalidade e arquitectura das aplicações que os usem.

Nomeadamente, pode observar-se como a concretização desta camada manutenção de vistas é independente das camadas inferiores, assentando apenas numa configuração particular dos aspectos de ordem e fiabilidade e não em operações dedicadas como existem em outras concretizações de protocolos em camadas, por exemplo, pela adição de operações de esvaziamento às camadas inferiores.

Finalmente fez-se uma breve análise do desempenho dos protocolos resultantes, realçando o facto dos dados obtidos não poderem ser directamente comparados com outros protocolos de grupo, uma vez que oferecem mais funcionalidade dentro dos protocolos e é feito intencionalmente um compromisso entre o desempenho na situação em que foi testado, de modo a poder obter um melhor desempenho em sistemas em grande escala.

Capítulo 8

Conclusões

8.1 Resumo das contribuições

No sentido de desenvolver protocolos de comunicação fiável que possam ser configurados para suportar eficientemente um largo espectro de aplicações, são nesta tese apresentados:

- a formalização do conceito de protocolo configurável como interpretador de especificações de protocolos;
- definição de uma linguagem capaz de descrever protocolos de difusão fiável.

Considerando cada uma das restrições às histórias dos processos que fazem parte da especificação de um protocolo como sendo frases de uma linguagem, apresenta-se um protocolo configurável como um interpretador dessa linguagem. Ou seja, define-se um protocolo configurável como sendo um algoritmo distribuído que cumpre um conjunto de restrições que recebe como parâmetro. Ao conjunto de todas as frases possíveis da linguagem chama-se o domínio do protocolo configurável.

Define-se então a linguagem adequada a descrever protocolos de difusão fiável como todas as frases no formato:

$$\forall p_j \in S, \rho(m, \dots) \Rightarrow \pi(H_j, m, \dots)$$

Cada uma destas frases impõe que os passos relativos a cada mensagem m que satisfaça a pré-condição $\rho(m, \dots)$ que aparecem nas histórias H_j de todos os p_j pertencentes a S, satisfaçam a restrição $\pi(H_j, m, \dots)$.

A definição das frases possíveis fica completa com a definição dos predicados ρ, com e sem acordo, e π, relativo a fiabilidade, ordem e vista. Deste modo definem-se seis tipos de frases que em conjunto com a definição de

S podem ser combinadas num grande número de especificações de protocolos adaptados à semântica das aplicações, pois cada mensagem pode estar sujeita a diferentes restrições.

Uma concretização de um protocolo configurável nestes termos tem necessariamente de suportar um número relativamente vasto de configurações, correspondendo às combinações dos vários tipos de frase para cada mensagem. Como tal, propõem-se soluções para os dois desafios fundamentais na concretização de um protocolo de comunicação configurável:

- conseguir uma modularização da concretização que permita reutilizar componentes de *software* entre diferentes configurações do protocolo, uma vez que o vasto número de combinações possíveis desaconselha concretizações separadas;

- proporcionar ao programador de aplicações uma interface adequada a exprimir a configuração de um protocolo, uma vez que a descrição da configuração escolhida passa a implicar um maior quantidade de informação enviada pela aplicação ao protocolo.

Como resposta ao segundo requisito, nesta tese propôs-se a abertura da concretização de protocolos de comunicação, através da reflexão das entidades relevantes à configuração do protocolo como objectos, permitindo a sua manipulação directa a partir do código da aplicação.

De modo a satisfazer o primeiro requisito, as entidades reflectidas como objectos de modo a serem directamente manipuláveis pela aplicação, foram os cabeçalhos individuais de cada mensagem e não o protocolo como um todo, como acontecia em anteriores concretizações abertas de sistemas distribuídos.

Deste modo, o comportamento do protocolo como um todo, adaptado à semântica da aplicação, é definido implicitamente ao ser escolhido um comportamento genérico a cada mensagem do mesmo modo que uma especificação é construída por um conjunto de restrições relativas a cada uma das mensagens.

8.2 Especificação de protocolos

A diferença fundamental em relação a anteriores especificações modulares de protocolos referidas no Capítulo 2 é não ser feita uma especificação de um protocolo como um todo, definindo assim o que é o comportamento correcto desse protocolo para com qualquer mensagem. Antes pelo contrário, é especificado o que é o comportamento correcto do protocolo individualmente para cada mensagem, o que resulta na definição implícita do protocolo como um todo.

A possibilidade de especificar o comportamento de um protocolo individualmente para cada mensagem durante a sessão de comunicação e não

apenas o comportamento do protocolo relativamente a todas as mensagens de uma sessão, é a característica que permite uma adaptação às necessidades da aplicação. Ou seja, esta capacidade é fundamental para permitir que a qualidade de serviço oferecida pelo protocolo varie em função da semântica das mensagens sem estar dependente de uma aplicação particular.

Para que um protocolo pudesse ser especificado como um todo e ser adaptável à semântica das aplicações então a especificação teria que ter em conta o conteúdo das mensagens. Se isso acontecesse, então qualquer concretização do protocolo deixaria de ser genérica. Evitando ter em conta o conteúdo das mensagens, são apenas possíveis para cada aspecto algumas categorias que resultam da aplicação de critérios uniformes a todas as mensagens.

Ao permitir que a especificação seja construída incrementalmente, esta pode referir-se às mensagens concretas trocadas e como tal ser adaptada a cada aplicação. No entanto, uma vez que cada incremento da especificação se refere à identidade da mensagem e não depende de juízos sobre o seu conteúdo, as frases podem ser genéricas para uma variedade de aplicações.

Isto resulta em que a caracterização de um protocolo configurável deixe de ser feita em termos de um conjunto de comportamentos alternativos mas sim em termos dos aspectos do seu comportamento que é possível alterar, ou seja, das frases de especificação que pode interpretar. Por exemplo, em vez se utilizarem as categorias apresentadas na Secção 2.4.2 para a ordenação, torna-se possível a configuração dos aspectos de ordem das entregas relativamente a cada um dos outros passos de difusão e entrega.

Como consequência, a utilização do protocolo resultante torna-se, para o programador de aplicações bastante mais complexa. Esta complexidade adicional não é no entanto um efeito secundário de uma concretização particular menos bem sucedida, mas é um facto incontornável da passagem de informação sobre a semântica das mensagens da aplicação para o sub-sistema de comunicação sob a forma da especificação incremental do protocolo.

Conclui-se então que um protocolo configurável é uma abstracção fundamentalmente diferente de um protocolo de comunicação em grupo, em que um dos aspectos centrais é a simplicidade do modelo oferecido, só possível com garantias fortes.

A utilização de protocolos configuráveis faz então sentido a dois níveis:

- em aplicações exigentes, como por exemplo sistemas distribuídos em grande escala, em que este esforço adicional é compensado por um desempenho superior;

- como base para sub-sistemas de comunicação de uso particular, como por exemplo, na concretização de modelos de coerência em sistemas de memória distribuída partilhada.

Em qualquer dos casos, pode reconhecer-se uma evolução de uma arqui-

tectura em dois níveis, formada pelo protocolo mais a aplicação, para uma arquitectura em três níveis, em que entre o protocolo configurável e a aplicação existe uma camada de adaptação, seja ela específica de uma aplicação em particular, ou genérica, por exemplo, sob a forma de um algoritmo de coerência de memória partilhada.

8.3 Concretização de protocolos

A concretização de protocolos cuja especificação é construída incrementalmente pela aplicação, é uma concretização aberta orientado por objectos. Neste contexto, concretização orientada por objectos de protocolos significa que:

- os protocolos manipulem objectos e não dados, pelo que a comunicação entre os dois extremos de uma sessão de comunicação efectuada normalmente através de cabeçalhos acrescentados à mensagem, passa a ser feita por troca de objectos que incluem as mensagens;
- uma vez que os cabeçalhos passam a ser objectos, possam ser utilizadas concretizações alternativas, desde que apresentem uma interface comum.

Uma concretização aberta implica ainda que as concretizações alternativas dos cabeçalhos possam ser seleccionadas pela aplicação, pelo que passa a ser responsabilidade desta a adição dos cabeçalhos às mensagens. Esta estratégia permite ultrapassar os dois problemas identificados, na medida em que permite à aplicação seleccionar a qualidade de serviço requerida para cada mensagem, ao mesmo tempo que:

- isso é feito reutilizando módulos desenvolvidos separadamente, correspondentes aos diferentes formatos de frases na especificação;
- a interface proporcionada à aplicação limita-se a primitivas simples para enviar e receber mensagens, sendo toda a sua complexidade visível apenas em proporção das necessidades da aplicação.

Neste contexto, a reutilização de módulos de *software* correspondentes aos cabeçalhos pode ser feita tanto por herança como por composição. Deste modo obtêm-se as vantagens inerentes a cada uma destas estratégias, em que a composição permite combinar módulos genéricos e independentemente reutilizáveis enquanto a herança é útil para obter componentes especializados para uma dada aplicação.

Apesar de não ser exigida, a migração de código pode ser utilizada como parte do mecanismo de migração de objectos, obtendo as vantagens inerentes às redes activas mas fornecendo uma arquitectura de alto nível que

ultrapasse as limitações das concretizações actuais que apenas exploram a mobilidade de código.

Uma conclusão importante é que a concretização aberta de protocolos pode ser utilizada com vantagens em qualquer tipo de protocolos, inclusivé em protocolos normalizados em termos de uma concretização tradicional. Para tal, basta definir camadas de serialização de objectos especializadas, que façam a conversão entre as representações externas normalizadas dos cabeçalhos de mensagens em representações internas baseadas em objectos adequadas a ser manipuladas pelos programadores de aplicações.

Uma questão fundamental na concretização aberta de protocolos orientados por objectos é a representação de histórias de processos, em particular, dos correspondentes aos passos de difusão e entrega de mensagens, uma vez que essa informação tem que ser manipulada pelos algoritmos e trocada como parte dos cabeçalhos. A solução utilizada normalmente para o efeito, representando os passos como números de sequência, é inadequada pois implica a exposição da concretização. A solução encontrada para este caso é um conjunto de componentes com interfaces bem definidas que permitem representar os diversos tipos de entidades que normalmente são necessárias.

Fazendo a comparação com técnicas existentes de modularização, como por exemplo a estratificação ou a composição de micro-protocolos, verifica-se que estas tornam difícil a especificação da qualidade de serviço desejada para cada mensagem. Por outro lado, a estratégia seguida pelas redes activas, que permite configurar a qualidade de serviço individualmente para cada mensagem, torna difícil essa configuração a partir da composição de módulos desenvolvidos em separado bem como a composição e inter-operação entre diferentes aspectos da configuração.

Como consequência, a possibilidade de configuração de um protocolo para cada mensagem implica a impossibilidade de reutilização de módulos ou então a manutenção de diversas sessões independentes configuradas em separado, o que impossibilita que sejam asseguradas garantias entre mensagens de sessões distintas.

Isto faz com que protocolos existentes que permitam a configuração da qualidade de serviço por cada mensagem sejam sobretudo protocolos monolíticos que oferecem um conjunto bem definido e fixo de opções acessíveis através de uma interface de programação relativamente complexa resultante da enumeração das opções aceites. A concretização de um protocolo configurável tal como o descrito usando esta estratégia, incorreria inevitavelmente nas dificuldades de manutenção e utilização inerentes a qualquer módulo de *software* complexo com uma interface de programação igualmente complexa.

8.4 Trabalho futuro

Como motivação para este trabalho, argumentou-se que a correcção e bom desempenho de aplicações tolerantes a faltas em redes em grande escala, passam pela adequação das garantias oferecidas pelo sistema de comunicação à semântica das aplicações.

No entanto o trabalho desenvolvido apenas demonstra que esse tipo de protocolos pode ser especificado em termos de propriedades genéricas e concretizado de uma forma eficiente como módulos de *software* reutilizáveis.

Como consequência, uma linha de investigação que agora pode e deve ser prosseguida é a demonstração que este tipo de protocolos são de facto vantajosos nas condições descritas. Para o efeito, terão que ser desenvolvidas aplicações sobre os protocolos configuráveis e testado seu desempenho sobre um sistema em grande escala, real ou simulado. Serão indicadas para este efeito, as aplicações apresentadas como exemplos de utilização nas diversas propostas de relaxamento de qualidade de serviço referidas no Capítulo 3.

Uma segunda vertente de investigação que pode ser explorada decorre da conclusão que um protocolo configurável constitui um paradigma de programação fundamentalmente diferente de protocolos não configuráveis, devido à complexidade acrescida que representa para o programador de aplicações ter que dar conhecimento ao protocolo da semântica das mensagens.

Para o efeito poder-se-á explorar outros paradigmas já existentes que melhor capturem essa semântica com um mínimo de esforço por parte do programador, como por exemplo, sistemas de memória partilhada distribuída de coerência fraca.

Outra alternativa, dada a interpretação da estratégia de configuração como introdução de reflexão das propriedades do protocolo como objectos, é a criação de *linguagens de aspectos* [KLM+97] correspondentes aos diferentes aspectos de configuração de protocolos especificados.

Uma terceira vertente de investigação é o melhoramento dos protocolos descritos e a concretização de outros protocolos. Por exemplo, para ultrapassar as limitações do protocolo simples baseado num algoritmo de consenso utilizado para a ordenação total. Ou então para a concretização de grupos hierárquicos, necessários para permitir sessões de comunicação com numerosos participantes.

Finalmente, faz ainda sentido efectuar um trabalho de optimização das infra-estruturas e dos protocolos existentes.

Bibliografia

[AF92] H. Attiya and R. Friedman. A correctness condition for high performance multiprocessors. In N. Alon, editor, *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 679–690, Victoria, B.C., Canada, May 1992. ACM Press.

[AFPS92] G. Agha, S. Frolund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *DCCA-3*, pages 197–207. Mondello, 1992.

[AMA+95] D. Agarwal, L. Moser, Y. Amir, P. Melliar-Smith, and R. Budhia. Reliable ordered delivery across interconnected local-area networks. In *Proceedings of the International Conference on Network Protocols*, pages 365–374, November 1995.

[Anc96] Emmanuelle Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems: proofs. Technical report, IRISA, November 1996.

[AOW96] E. Arjomandi, W. O'Farrell, and G. V. Wilson. Smart messages: An object-oriented communication mechanism. In USENIX Association, editor, *2nd Conference on Object-Oriented Technologies & Systems (COOTS), June 17–21, 1996. Toronto, Canada*, pages 233–240, Berkeley, CA, USA, June 1996. USENIX.

[AP93] M. Abbott and L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, February 1993.

[BBD96] Ö. Babaoğlu, A. Bartoli, and G. Dini. Enriched view sinchrony: A paradigm for programming dependable applications in partitionable asynchronous distributed systems. Technical Report UBLCS-96-3, Dept. of Computer Science, University of Bologna, February 1996.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[BCBT96] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of WDAG'96*, 1996.

[BDGB95] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 2: Software Technology*, pages 612–621, Los Alamitos, CA, USA, January 1995. IEEE Computer Society Press.

[BDM95] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications. Technical report, BROADCAST, 1995. UBLCS-95-18.

[BFvR96] R. Baldoni, R. Friedman, and R. van Renesse. The hierarchical daisy architecture for causal delivery. Technical Report TR96-1610, Cornell University, Computer Science Department, September 1996.

[BG93] K. Birman and B. Glade. Consistent Failure Reporting in Reliable Communication Systems. Technical Report 93-1349, Department of Computer Science, Cornell University, May 1993.

[Bir93] Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 1993.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138, Austin, TX, November 1987. In *ACM Operating Systems Review 21*:5.

[BMST93] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In Sape Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison Wesley, second edition, 1993.

[BO93] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1993.

[BS94] Ö. Babaoğlu and A. Schiper. On group communication in large-scale distributed systems. In *Proceedings of the 6th SIGOPS European Workshop*, September 1994.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272, August 1991.

[CHT94] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. Technical report, Department of Computer Science, Cornell University, May 1994.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[CS97] O. Ciupke and R. Schmidt. Components as context-independent units of software. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming*, pages 139–143. dpunkt-Verlag, 1997.

[CT94] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical report, Department of Computer Science, Cornell University, 1994.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):288–323, April 1987.

[Dee88] Stephen Deering. *RFC1112: Host Extensions for IP Multicasting.* SRI Network Information Center, August 1988.

[DLS88] D. Dolev, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):347–382, April 1988.

[DM96] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[DMS96] D. Dolev, D. Malki, and R. Strong. A framework for partitionable membership service. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 343–343, New York, May 1996. ACM.

[DZ83] J. Day and H. Zimmermann. The OSI reference model. *Proc. of the IEEE*, 1983.

[EMS95] P. Ezhilchelvan, R. Macêdo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 296–306, Los Alamitos, CA, USA, May 1995. IEEE Computer Society Press.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 1985.

[FNP+95] J-C. Fabre, V. Nicomette, T. Perennou, R. Stroud, and Z. Wu. Implementing fault-tolerant applications using reflective object-oriented programming. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pages 489–498, Pasadena, California, 1995.

[Fon94] Henrique Fonseca. Ambientes de suporte para modularização, concretização e execução de protocolos de comunicação. Master's thesis, Universidade Técnica de Lisboa, Instituto Superior Técnico, 1994.

[Fri95] Roy Friedman. Using virtual synchrony to develop efficient fault tolerant distributed shared memories. Technical Report TR-95-1506, Dept. of Computer Science, Cornell University, March 1995.

[GFG96a] B. Garbinato, P. Felber, and R. Guerraoui. Protocol classes for designing reliable distributed environments. In P. Cointe, editor, *Proceedings ECOOP '96*, LNCS 1098, pages 316–343, Linz, Austria, July 1996. Springer-Verlag.

[GFG96b] B. Garbinato, P. Fleber, and R. Guerraoui. Using the Strategy pattern to compose reliable distributed protocols. In *Proceedings of the 3rd Conference on the Patterns Languages of Programs (PLoP'96)*, September 1996.

[GFG97] B. Garbinato, P. Felber, and R. Guerraoui. Modeling protocols as objects for structuring reliable distributed systems. In *Proceedings of the Communications Networks and Distributed Systems Modeling and Simulation Conference (CNDS'97)*, January 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.

[GOS96] R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical report, LSE, EPF Lausanne, December 1996.

[GS95] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony models: Bridging the gap. *Lecture Notes in Computer Science*, 938:121–131, 1995.

[GS96a] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building agreement protocols in distributed systems. In *Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing*, pages 168–177, Washington, June 1996. IEEE.

[GS96b] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. *Lecture Notes in Computer Science*, 1088, 1996.

[GvRVB97] K. Guo, R. van Renesse, W. Vogels, and K. Birman. Hierarchical message stability tracking protocols (abstract). Cornell University, Computer Science Department, 97.

[HB96] M. Hayden and K. Birman. Probabilistic broadcast. Technical report, Computer Science Department, Cornell University, September 1996. TR96-1606.

[Hil96] Matti Hiltunen. *Configurable Fault-Tolerant Distributed Services*. PhD thesis, Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, July 1996.

[HJE95] Hermann Hueni, Ralph E. Johnson, and Robert Engel. A framework for network protocol software. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, October 1995.

[HP91] N. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HS95] M. Hiltunen and R. Schilchting. Understanding membership. Technical Report TR95-07, Dept. of Computer Science, University of Arizona, 1995.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison Wesley, second edition, 1993.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Computer Science Department, Cornell University, 1994.

[HW90] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[IP94] D. Ingham and G. Parrington. Delayline: A wide-area network emulation tool. In USENIX Association, editor, *Computing Systems, Summer, 1994.*, volume 7, pages 313–332, Berkeley, CA, USA, Summer 1994. USENIX.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[KG97] A. Krall and R. Graft. CACAO - A 64 bit JavaVM just-in-time compiler. PPoPP'97 Workshop on Java for Science and Engineering Computation, 1997.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 Object Oriented Programming, Proceedings*, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, 1997.

[KP96] G. Kiczales and Xerox Parc. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, January 1996.

[KV93] H. Kopetz and P. Veríssimo. Real time and dependability concepts. In Sape Mullender, editor, *Distributed Systems*, chapter 16, pages 411–446. Addison Wesley, second edition, 1993.

[Lap92] J. C. Laprie. *Dependability: Basic Concepts and Terminology.* Springer Verlag, 1992.

[lB97] A. Ólafsson and D. Bryan. On the need for "required interfaces" of components. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming*, pages 159–165. dpunkt-Verlag, 1997.

[LLS91] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. *ACM SIGOPS Operating Systems Review*, 25(1):49–54, January 1991.

[LY96a] T. Lindholm and F. Yellin. *The Java Application Programming Interface: Core Packages*, volume 1 of *The Java Series*. Addison-Wesley, 1996.

[LY96b] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.

[Mal95] Christoph Malloth. Increasing reliability of communication in large scale distributed systems. In *IASTED 7th Intl. Conf. on Parallel and Distributed Computing and Systems*, October 1995.

[Mal96] Christoph Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, Swiss Federal Institute of Technology of Lausanne (Ecole Polytechnique Fédérale de Lausanne), September 1996.

[MAMSA94] L. Moser, Y. Amir, P. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[MBM95] S. Maffeis, W. Bischofberger, and K. Mätzel. A generic multicast transport service to support disconnected operation. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, 1995.

[McA95] Jeff McAffer. Meta-level programming with CodA. In Walter Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *LNCS*, pages 190–214, Berlin, GER, August 1995. Springer.

[MMBC97] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In USENIX, editor, *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 16–19, 1997. Portland, Oregon*, pages 1–20, Berkeley, CA, USA, June 1997. USENIX.

[MMSA+96] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast

group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[MMT94] M. Muhugusa, G. Marzo, and C. Tschudin. ComScript: An environment for the implementation of protocol stacks and their dynamic reconfiguration. In *Proceedings of ISACC'94*, Monterrey, Mexico, 1994.

[MMTH95] G. Di Marzo, M. Muhugusa, C. Tschudin, and J. Harms. The messenger paradigm and its implications on distributed systems. In *Proceedings of ICC'95 Workshop on Intelligent Computer Communication*, 1995.

[MR93] A. Mostefaoui and M. Raynal. Causal multicasts in overlapping groups: Towards a low cost approach. In *Proceedings of the IEEE Int. Conference on Future Trends in Distributed Computing Systems*, September 1993.

[MR94] A. Mostefaoui and M. Raynal. Definition and implementation of a flexible communication primitive for distributed computing. In *Proceedings of the IFIP WG 10.3 International Conference on Applications of Parallel and Distributed Computing.* North-Holland, April 1994.

[Obj93] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, December 1993. Revision 1.2.

[Obj94] Object Management Group. *Common Object Services Specification, Volume I*, March 1994. Revision 1.0.

[OGS96] R. Oliveira, R. Guerraoui, and A. Schiper. Solving consensus in the presence of omission failures. Technical report, EPFL, 1996.

[OP92] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[PBS89] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[PS94] F. Panzieri and S. Shrivastava. A View of Large Scale Distributed Computing. In *Broadcast Project Deliverable Report*, October 1994.

[PSWL95] G. Parrington, S. Shrivastava, S. Wheater, and M. Little. The design and implementation of Arjuna. In USENIX, editor, *Computing Systems, Summer, 1995.*, volume 8, pages 255–308, Berkeley, CA, USA, Summer 1995. USENIX.

[RFV96] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *ICDCS '96; Proceedings of the 16th International Conference on Distributed Computing Systems; May 27-30, 1996, Hong Kong*, pages 503–510, Washington - Brussels - Tokyo, May 1996. IEEE.

[RM93] M. Raynal and M. Mizuno. How to find his way in the jungle of consistency criteria for distributed shared memories (or how to escape from minos' labyrinth). In *Proc. of the IEEE Int. Conf. on Future Trends of Distributed Computing Systems*, pages 340–346, Lisboa (Portugal), September 1993.

[RV92] L. Rodrigues and P. Veríssimo. xAMp: A multi-primitive group communication service. In *Proceedings of the 11th Symposium On Reliable Distributed Systems*, October 1992.

[RV94] L. Rodrigues and P. Veríssimo. How to avoid the cost of causal communication in large-scale systems. In *Proceedings of the 6th SIGOPS European Workshop*, September 1994.

[RV95] L. Rodrigues and P. Veríssimo. Causal separators and topological timestamping: An approach to support causal multicast in large-scale systems. Technical report, Technical University of Lisboa, IST - INESC, 1995.

[RWWB96] R. Riggs, J. Waldo, A. Wollrath, and K. Bharath. Pickling state in the Java system. *Usenix Computing Systems*, 9(4):291–312, Fall 1996.

[SBS93] D. Schmidt, D. Box, and T. Suda. ADAPTIVE - A dynamically assembled protocol transformation, integration and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, June 1993.

[Sch93a] Douglas Schmidt. The ADAPTIVE communications environment. In *Proceedings of the 11th Sun User Group Conference and Exhibition*, pages 214–225, Brookline, MA, USA, December 1993. Sun User Group, Inc.

[Sch93b] Fred Schneider. Replication management using the state-machine approach. In Sape Mullender, editor, *Distributed Systems*, chapter 7, pages 169–197. Addison Wesley, second edition, 1993.

[Sch97] André Schiper. Early consensus in an asynchronous system with a weak failure detector. *DISTCOMP: Distributed Computing*, 10, 1997.

[SFG+96] J. Smith, D. Farbert, C. Gunter, S. Nettles, D. Feldneier, and W. Sincoskie. SwitchWare: Accelerating network evolution. Technical report, CIS Department, University of Pennsylvania, 1996. MS-CIS-96-38.

[SG97] André Schiper and Rashid Guerraoui. Total order multicast to multiple groups. In *Proceedings of the 17th Int. Conference on Distributed Computing Systems*, pages 578–585, May 1997.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *distrcomp (?)*, 7(3), March 1994.

[SP97] C. Szyperski and C. Pfister. Component-oriented programming: WCOP'96 Workshop report. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming*, pages 127–130. dpunkt-Verlag, 1997.

[SR93] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. Technical report, Cornell University, Computer Science Department, 1993. TR93-1993.

[Str91] Bjarne Stroustrup. *The C++ Programming Language: Second Edition.* Addison-Wesley Publishing Co., Reading, Mass., 1991.

[Sun96a] Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043. *Java Object Serialization Specification*, December 1996. 1.2.

[Sun96b] Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043. *JavaSoft JavaBeans*, December 1996. 1.00-A.

[TMMH94] C. Tschudin, G. Di Marzo, M. Muhugusa, and J. Harms. Messenger-based operating systems. *Cahier du Centre Universitaire d'Informatique*, July 1994. Revised September, 1994.

[TSS+97] D. Tannenhouse, J. Smith, W Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications*, 35(1):80–86, January 1997.

[TW96] D. Tannenhouse and D. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), April 1996.

[vR96] Robbert van Renesse. The Horus uniform group interface. Technical report, Cornell University, 1996.

[vRB95] R. van Renesse and K. Birman. Protocol composition in Horus. Technical Report TR95-1505, Cornell University, Computer Science Department, March 1995.

[VRV92] P. Veríssimo, L. Rodrigues, and W. Vogels. Group orientation: A paradigm for modern distributed systems. In *Proceedings of the ACM SIGOPS 1992 Workshop*, 1992.

[VVR94] P. Veríssimo, W. Vogels, and L. Rodrigues. A framework for structuring group support in lsdcs. In *Broadcast Project Deliverable Report*, October 1994.

[WT96] D. Wetherall and D. Tennenhouse. The Active IP option. In *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

Apêndice A

Programação de componentes

A.1 Infra-estrutura de componentes

A.1.1 Introdução

Esta seccão apresenta uma introdução à programação com a infra-estrutura de componentes e a linguagem JAVA, tal como descrita na Secção 6.3.

Para o efeito é a apresentada a especialização da infra-estrutura genérica numa infra-estrutura de grafos de fluxo de dados. Este tipo de construção é a base para a agregação de protocolos de comunicação simples, generalizando a composição de protocolos por camadas empilhadas.

A.1.2 Fluxo de dados

A agregação de componentes de protocolos de comunicação é feita com base no fluxo de dados. Sendo os dados em geral definidos como entidades discretas representadas como objectos, a especificação da aceitação de um fluxo de dados por parte de um componente consiste na disponibilização de um serviço do tipo `DataFlow` definido na Figura A.1.

Esta definição é semelhante à definição de `PushConsumer` no serviço de comunicação de eventos [Obj94] da norma CORBA [Obj93], tendo uma semântica idêntica.

A.1.3 Componentes simples

Um componente simples é qualquer objecto válido em JAVA. No entanto recomenda-se que para manter as vantagens de um estrutura de componentes:

```
public interface DataFlow {
    public void push(Object data);
};
```

Figura A.1: Definição do tipo das características de fluxo de dados.

```
public class Sink implements DataFlow {
    public void push(Object data) {
        System.out.println("*puf* "+data+" removido.");
    }
};
```

Figura A.2: Um componente simples com apenas um serviço.

- todas as estruturas de dados sejam privadas;
- todas as operações que são públicas sejam definidas em especificações de interface separadas e não implicitamente pelo componente;
- todas as referências sejam feitas com os tipos dos serviços e não directamente aos componentes que os concretizam;
- todas as referências a outros componentes, sobretudo se não forem constantes e puderem existir mais referências para eles, sejam feitas através de dependências externas explícitas.

A Figura A.2 apresenta um componente que concretiza um serviço de fluxo de dados, limitando-se a imprimir os dados.

A.2 Dependências externas

Cada dependência externa é apresentada como um serviço do componente, que permite que o serviço referido por essa dependência seja alterado. Este serviço é especificado por `Dependency` (Figura A.3) e é constituído por operações para modificar e consultar o serviço alvo da ligação.

A utilização de nomes para os métodos segundo a norma definida para componentes JavaBeans [Sun96b], neste caso de um atributo modificável, faz com que as duas infra-estruturas possam ser compatibilizadas com facilidade.

A Figura A.4 apresenta um componente simples com um serviço e uma dependência externa como características. Este componente pode ser usado como um componente que não altera um fluxo de dados unidireccional, ou como um componente que inverte um fluxo de dados bidireccional, demonstrando a flexibilidade da infra-estrutura de componentes ao permitir

```
public interface Dependency {
    public void setTarget(Object target);

    public Object getTarget();
};
```

Figura A.3: Especificação do serviço que concretiza uma dependência externa de um componente.

```
public class Loopback implements Dependency, DataFlow {
    public void setTarget(Object target) {
        this.target=(DataFlow)target;
    }

    public Object getTarget() {
        return target;
    }

    public void push(Object data) {
        target.push(data);
    }

    private DataFlow target;
};
```

Figura A.4: Um componente simples com um serviço e um dependência externa.

utilizar a mesma interface sempre que possível, por oposição a uma camada de protocolo tradicional onde o topo de uma camada só pode ser ligado à base de outra e vice-versa, embora a semântica do topo e da base, num fluxo bidireccional seja estritamente idêntica.

A.2.1 Componentes dinâmicos

Um componente dinâmico permite definir as suas características durante a execução do programa. Para o efeito, disponibiliza um serviço que traduz uma etiqueta que refere o serviço para um referência para esse serviço (Figura A.5).

Como a concretização deste serviço é da exclusiva responsabilidade do componente, pode ser efectuada de vários modos:

- por comparações sucessivas, quando é um número fixo e reduzido de

```
public interface DynamicComponent {
    public Object getFeature(Object label);
};
```

Figura A.5: Serviço de resolução de características para componentes dinâmicos.

características;

- armazenando os pares etiqueta-característica num tabela, quando é um número grande ou variável de caracteristicas, como é utilizado no componente abstracto `GenericComponent`;
- um combinação destas estratégias, nomeadamente por especializações sucessivas por herança.

A Figura A.6 um componente que serve para filtrar um dos sentidos de um fluxo bidireccional de dados segundo uma condição parameterizável, em que:

- se usa o componente `GenericComponent` como base, de modo a herdar um concretização genérica do serviço `DynamicComponent` baseada em tabelas;
- se usam objectos da classe `Key` que permite definir identificadores globalmente únicos no espaço de nomes de um componente, previnindo colisões com nomes de sub-classes e super-classes;
- se usa a classe `PersistentDependency`, que concretiza uma dependência de fluxo de dados, como base para características;
- se demonstra a utilidade das definições de classes aninhadas como "açucar sintático" para definir as características de um componente dinâmico;
- se exemplifica uma dependência, e em consequência, uma característica estática num componente dinâmico.

De modo a simplificar o tratamento de componentes simples e dinâmicos, podem ser utilizadas as funções `Get.feature` e `Get.dependency` para obter de um componente uma referência para uma característica. No caso de um componente dinâmico, estas funções invocam o serviço apropriado no componente. Caso contrário, a etiqueta é ignorada e é devolvida a própria referência para o componente que lógicamente é uma referência para todas as suas características.

```
public class Filter extends GenericComponent implements Dependency {
    public static final Object In=new Key("groupz.flow.Filter", 0);
    public static final Object Out=new Key("groupz.flow.Filter", 1);

    public Filter() {
        class InFeature extends PersistentDependency
          implements DataFlow {
            public void push(Object data) {
                if (condition.test(data))
                    out.pop(data);
            }
        };
        class OutFeature extends PersistentDependency
          implements DataFlow {
            public void push(Object data) {
                in.pop(data);
            }
        };
        addFeature(In, in=new InFeature());
        addFeature(Out, out=new OutFeature());
    }

    public void setTarget(Object target) {
        condition=(Condition)target;
    }

    public Object getTarget() {
        return target;
    }

    private PersistentDependency out, in;
    private Condition condition;
};
```

Figura A.6: Um componente com dois serviços e duas dependências.

```
public abstract class Get {
    public static Object feature(Object label, Object obj);

    public static Dependency dependency(Object label, Object obj);
};
```

Figura A.7: Funções utilitárias para obter de um componente, uma qualquer característica ou uma dependência externa.

A.2.2 Grafos estáticos

Os componentes individuais podem então ser combinados em grados, satisfazendo as dependências com serviços adequados. Para o efeito pode ser utilizado o componente `StaticGraph` que permite inicializar um grafo com um sequência de operações de ligação de dependências a serviços. Estas ligações podem ser unidireccionais ou bidireccionais. A Figura A.8 demonstra a utilização deste tipo de componente para configurar um grafo esquematizado na Figura A.9. A Figura A.10 demonstra o resultado da execução do componente formado.

A.3 Infra-estrutura de sincronização

A.3.1 Introdução

A infra-estrutura de histórias apresenta para cada tipo de entidade representada, dois tipo de componentes:

- um componente que actua como um conjunto ao qual podem ser acresentaods elementos e que pode ser interrogado quando ao seu conteúdo;
- um conjunto de componentes que depois de criados são imutáveis e que actuam como actualizações ou interrogações sobre o conjunto.

Um elemento chave para a o sucesso desta estrutura é a possibilidade de efectuar diferentes tipos de consulta de inclusão de conjuntos, nomeadamente, é importante a possibilidade de bloquear actividades até uma dessas condições ser verdadeira.

Esta secção apresenta o mecanismo utilizado para efectuar a sincronização, de modo a que as diversas classes possam ser compostas arbitrariamente em condicões complexas.

A.3.2 Sincronização

Por motivos de eficiência, estes componentes apenas permitem tratar condições que são inicialmente falsas, num dado momento passam a verdadeiras

```
public class Teste extends StaticGraph {
    public static final Object In="In";

    public Teste() {
        Object filter=new Filter();
        Object cond=new Condition() {
            public boolean test(Object obj) {
                System.out.println("Testando "+obj+"...");
                return obj instanceof String &&
                       ((String)obj).startsWith("Teste");
            }
        };
        Object loop=new Loopback();
        Object sink=new Sink();

        export(filter, Filter.In, In);
        link(filter, Filter.In, sink, null);
        link(filter, null, cond, null);
        bilink(filter, Filter.Out, loop, null);
    }

    public static void main(String args[]) {
        DataFlow in=(DataFlow)Get.feature(Teste.In, new Teste());
        for(int i=0;i<args.length;i++)
            in.push(args[i]);
    }
};
```

Figura A.8: Programa baseado num componente composto.

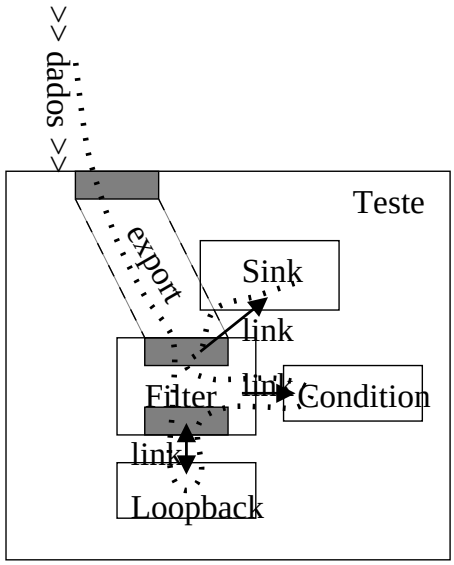


Figura A.9: Diagrama de estrutura do componente `Teste`.

```
$ java groupz.demo.Teste traste Testes toste
Testando traste...
Testando Testes...
*puf* Testes removido.
Testando toste...
```

Figura A.10: Resultado de uma execução do componente `Teste`.

e assim se mantêm. Uma transição de verdadeiro para falso é pois considerada incorrecta. Esta limitação é no entanto desprezável para tratar condições de inclusão de elementos em conjuntos estritamente crescentes, como é o caso na infra-estrutura de representação de histórias.

Em relação aos mecanismos básicos de espera e notificação existentes na linguagem JAVA, estes compoenentes tornam possível a composição de condições de espera em operações de disjunção, necessárias, por exemplo, para o espera por maiorias. Ao mesmo tempo proporcionam a base para as facilidades de diagnóstico.

Um componente que concretize uma condição pela qual clientes possam aguardar deve concretizar o serviço `WaitServer` (Figura A.11). A notificação é feita através do serviço descrito na Figura A.13, desbloquando todas os clientes em espera síncrona e notificando todos os clientes em espera assíncrona.

Um cliente pode esperar que uma condição seja verdadeira:

- utilizando o método de espera síncrono que bloqueia a actividade até

```
public interface WaitServer {
    // Espera síncrona
    public void synchronousWait() throws InterruptedException;

    // Espera assíncrona
    public void addAsynchClient(AsynchWaitClient cli, WaitServer ws);

    public boolean test();

    public void removeAsynchClient(AsynchWaitClient cli);
};
```

Figura A.11: Descrição de serviços para espera síncrona ou assíncrona.

```
public interface AsynchWaitClient {
    // Notificação assíncrona
    public void notification(WaitServer srv);
};
```

Figura A.12: Descrição de serviço para cliente de espera assíncrona.

a condição ser verdadeira;

- concretizando o serviço descrito na Figura A.12 e registando-se utilizando os métodos para espera assíncrona.

É então disponibilizado um conjunto de componentes utilitários que concretizam estes serviços:

- um componente correspondente a uma condição atómica que é notificada do exterior;
- componentes que concretizam as operações lógicas de conjunção e disjunção sobre um qualquer conjunto de outros componentes;
- um componente que concretiza a condição de uma maioria entre um conjunto de quaisquer outros componentes ser verdade.

Estes componentes podem então ser combinados para efectuar consultas de representações de histórias.

A.3.3 Diagnóstico

Estão ainda disponíveis dois componentes que podem ser usados para:

```
public interface NotificationServer {
    // Notificação
    public void notifyAllClients();
};
```

Figura A.13: Descrição do serviço para notificação.

- etiquetar uma qualquer condição com um objecto, que pode ser considerada a identificação do "culpado" por a condição correspondente ser falsa;
- reunir informação sobre quais os culpados do bloqueio de actividades.

Estes últimos são usados, por exemplo, para a averiguação de condições que levam ao bloqueio de actividades de entrega de mensagens e consequentemente a despoletar mudanças de vista.

Apêndice B

Programação de protocolos

B.1 Introdução

Este capítulo apresenta alguns fragmentos de código que ilustram a utilização do GROUPZ para difusão de mensagens com diversas qualidades de serviço. São no entanto apenas exploradas algumas das possibilidades dos carregadores genéricos, não sendo portanto tomada em consideração a possbilidade de escrita de carrgadores específicos.

Em todos estes exemplos assume-se que a variável `ch` representa um serviço de fluxo de dados correspodendente a uma pilha de protocolos apropriada. Considera-se ainda a existência de uma variável inteira `i` que funciona como um contador e uma colecção `view` que enumera os processos destinatários das mensagens enviadas. A variável `data` contém o objecto que constitui a mensagem a ser difundida.

B.2 Fiabilidade

Um carregador da camada de fiabilidade especifica como parâmetros, respectivamente:

- um carregador para a camada inferior[1];
- uma identificação para a mensagem;
- uma confirmação da recepção de outras mensagens;
- uma condição de entrega;
- uma condição de terminação.

[1] Em todas as configurações apresentadas este carregador é `null` uma vez que a camada de fiabilidade é a camada mais baixa com concretização aberta.

```
Event c=new OneEvent(oid, new SeqRange(i++));
Reception conf=new AllReceptions(view, c);

Event id=new OneEvent(oid, new SeqRange(i));
Reception term=new AllReceptions(view, id);

GenericMulticast mc=new ReliableMulticast(null, id, conf, null, term);

ch.push(mc.carry(data));
```

Figura B.1: Uma mensagem num canal insistente.

```
Event id=new OneEvent(oid, new OneSeq(i++));
Reception idtoall=new AllReceptions(view, id);
Reception idtomaj=new MajReceptions(view, id);

GenericMulticast mc=new ReliableMulticast(null, id, null,
                                          idtomaj, idtoall);

ch.push(mc.carry(data));
```

Figura B.2: Uma mensagem de entrega atómica e fiável.

A Figura B.1 apresenta a difusão de uma mensagem num canal insistente[2]. Neste caso, cada mensagem tem como identificação toda a sequência de mensagens produzidas pelo mesmo processo. Cada mensagem confirma ainda a recepção de todas as suas predecessoras, o que na prática termina a sua retransmissão.

A Figura B.2 apresenta a difusão de uma mensagem fiável e atómica, ou seja, invitávelmente entregue a todos os processos correctos se for entregue por algum processo, correcto ou não. Para o efeito é especificada como condição prévia à entrega, a reunião de uma maioria de confirmações de recepção.

B.3 Ordem

O carregador correspondente à camada de entrega ordenada recebe como parâmetros, respectivamente:

- um carregador para a camada inferior responsável pela entrega fiável;

[2]Ver Secção 2.5.2.

```
Event antes=new OneEvent(oid, new SeqRange(i++));
Event depois=new OneEvent(oid, new OneSeq(i));

mc=new OrderMulticast(mc, null, depois, antes);

ch.push(mc.carry(data));
```

Figura B.3: Ordem FIFO.

```
// Estado global
EventSet antes=new EventSet();

// Durante a recepcao
antes.put(id);

// Durante a difusao
Event id=new OneEvent(oid, new OneSeq(++i));

mc=new OrderMulticast(mc, null, id, antes);

antes.put(id);

ch.push(mc.carry(data));
```

Figura B.4: Ordem causal.

- um conjunto de fragmentos de história futura, para ordenação em relação outras entregas;
- um fragmento de história local correspondendo à identificação da mensagem;
- um conjunto de fragmentos de história local, para ordenação em relação à difusão.

A difusão de uma mensagem com ordenação FIFO é apresentada na Figura B.3. Neste caso, não é utilizada nenhuma dependência em relação a outra entrega, apenas um dependência em relação a todas as mensagens préviamente emitidas pelo mesmo processo.

A ordenação causal, apresentada na Figura B.4 implica que a identificação de todas as mensagens, tanto difundias como entregues seja reunida, o que é feito em `antes`. Esse conjunto de fragmentos de história local é então utilizado como dependência de ordem para a mensagem difundida.

```
Event id=new OneEvent(oid, new OneSeq(++i));

Container ids=new VectorContainer();
ids.addElement(new FutureEvent("cookie", true));

mc=new OrderMulticast(mc, ids, id, null)

ch.push(mc.carry(data));
```

Figura B.5: Ordem total.

Finalmente, a Figura B.5, apresenta a difusão de uma mensagem totalmente ordenada. Para o efeito, é utilizado como parâmetro um fragmento de história futura criado com um identificador arbitrário e um valor lógico verdadeiro.

B.4 Gestão do grupo

O carregador correspondente à camada manutenção da vista do grupo recebe como parâmetros, respectivamente:

- um carregador para a camada inferior responsável pela entrega ordenada;
- dependências de difusão e entrega a adicionar;
- dependências de difusão e entrega a remover;
- uma fábrica para um carregador de notificação.

Uma difusão de uma mensagem em sincronismo virtual significa que essa mensagem será entregue na mesma vista em que foi difundida ou previamente entregue, tal como é apresentado na Figura B.6. A utilização de dependências de entrega a nível da camada de vistas implica que seja incluída uma dependência de ordem correspondente.

Uma outra configuração possível apresentada na Figura B.7 é restringir as dependências de vista apenas à manutenção do mesmo líder da vista de difusão. Neste caso, não é necessária qualquer modificação nas dependências de ordem.

```
Event id=new OneEvent(oid, new OneSeq(++i));

Container ids=new VectorContainer();
ids.addElement(new FutureEvent("view", false));

mc=new OrderMulticast(mc, ids, id, null)

Container md=new VectorContainer();
md.addElement(new SimpleMulticastDependency(view, id));
Container dd=new VectorContainer();
dd.addElement(new SimpleDeliveryDependency(view,
                       new FutureEvent("view", true)));

mc=new MembershipMulticast(mc, md, dd, null, null, null);

ch.push(mc.carry(data));
```

Figura B.6: Entrega garantida na mesma vista da difusão e de outras entregas.

```
Event id=new OneEvent(oid, new OneSeq(++i));

mc=new OrderMulticast(mc, null, id, null)

Container md=new VectorContainer();
md.addElement(new LeaderMulticastDependency(view, id));

mc=new MembershipMulticast(mc, md, null, null, null, null);

ch.push(mc.carry(data));
```

Figura B.7: Entrega garantida numa vista com o mesmo lider da difusão.

Apêndice C

Algoritmo de consenso

C.1 Algoritmo

O problema do consenso, tal como descrito na Secção 2.5, é uma formulação abstracta do problema de acordo em sistemas distribuídos.

O GROUPZ usa uma algoritmo de consenso baseado em difusão [Sch97] que na ausência de falhas permite chegar a uma decisão com apenas duas trocas de mensagens.

Adicionalmente, esta concretização assume que todos os processos correctos efectuam um número ilimitado de iterações do algoritmo de consenso, cumprindo a especificação do serviço de consenso assumida como parte do modelo na Secção 5.2.6.

Embora esta possibilidade não tenha sido explorada, tal como acontece com os restantes protocolos descritos no Capítulo 7 esta concretização é parametrizável, pois os carregadores utilizados dependem todos do carregador utilizado por cada processo para fazer a proposta inicial, que pode ser definido pela aplicação.

C.2 Hospedeiro

O hospedeiro para acordo (Figura C.1) disponibiliza serviços para:

- identificar qual o conjunto de processos que participa actualmente no problema;
- o identificador do processo correspondente ao hospedeiro;
- o número identificador da ronda, o que permite por exemplo calcular qual o coordenador correspondente;
- estimativa actual por parte do processo, incluíndo qual a ronda e o processo que a estabeleceram;

```
hospedeiro Consenso exporta
                    P,   /* identificadores dos processos */
                    i,   /* identificador do processo local */
                    r_i, /* número da ronda */
                    e_i, /* (v, j, r_j) estimativa */
  conf, canc, susp,  /* conjuntos de identificadores de processos */
          decidido_n,  /* verdade se a iteração n foi decidida */
                    D   /* conjunto de processos suspeitos */
fim
```

Figura C.1: Componente hospedeiro para consenso.

```
carregador Proposta para o hospedeiro Consenso usa
  t,  /* iteração */
  v,  /* proposta inicial */
 P_t, /* conjunto de processos participantes */
  j   /* identificador do processo proponente */
faz
  e_i ← (v, i, 0);
  P ← P_t;
  i ← j;
  inicia Coordenador(t);
  inicia Suspeitas(t);
  espera por decidido_t;
  entrega valor(e_i);
fim
```

Figura C.2: Componente carregador para iniciar uma iteração do consenso.

- conjuntos para reuinir informação relativa ao estado dos restantes processos;
- informação sobre a iteração do probelema;
- um serviço para consulta de um detector de falhas.

C.3 Carregadores

Uma iteração de consenso é iniciada pela emissão por cada um dos processos participantes de um carregador da proposta para o seu hospedeiro local. Este carregador (Figura C.2) acompanha todo o desenrolar da iteração:

```
carregador Coordenador para o hospedeiro Consenso usa
 t   /* iteração */
faz
   enquanto ¬decidido_t faz
      espera por coord(r_i) = i ∨ decidido_t;
      se ¬decidido_t então
         r ← r_i;
         e_i ← (valor(e_i), i, r_i);
         envia Estimativa(t, e_i, i);
         espera por r_i > r ∨ decidido_t;
      fim
   fim
fim
```

Figura C.3: Componente carregador para desempenhar o papel de coordenador.

- estabelecendo a estimativa inicial, o grupo de processos participantes e a identificação do processo local;
- iniciando uma actividade paralela para desempenhar o papel de coordenador;
- iniciando outra actividade paralela para vigiar o detector de falhas, de modo a emitir suspeitas;
- esperando que seja conhecida uma decisão, que entrega à camada superior.

Enquanto não é atingida uma decisão, o carregador coordenador (Figura C.3) espera pelas rondas em que o processo local deve assumir o papel de coordenador, caso em que:

- marca sua estimativa actual como a estimativa do coordenador da ronda corrente;
- envia a sua estimativa a todos os processos;
- espera pela terminação da ronda ou pela decisão.

Durante o decorrer de qualquer ronda, um carregador (Figura C.4) vigia o detector de falhas local, à espera que o coordenador dessa ronda seja suspeitado. Se isso acontecer, então essa suspeita é comunicada a todos os outros processos através de um carregador da suspeita (Figura C.4).

```
carregador Suspeitas para o hospedeiro Consenso usa
 t   /* iteração */
faz
   enquanto ¬decidido_t faz
      espera por coord(r_i) ∈ D ∨ decidido_t;
      se ¬decidido_t então
         r ← r_i;
         envia Suspeita(t, i, r_i);
         espera por r_i > r ∨ decidido_t;
      fim
   fim
fim
```

Figura C.4: Componente carregador para emitir suspeitas.

```
carregador Estimativa para o hospedeiro Consenso usa
  t,   /* iteração */
 e_j,  /* estimativa */
  j    /* processo emissor da estimativa */
faz
   espera por decidido_{t-1};
   espera por r_i =;
   se ¬decidido_t ∧ canc = ∅ então
      conf ← conf ∪ {j};
      se |conf| = 1 então
         e_i ← e_j;
         se i ≠ coord(r_i) então
            envia Estimativa(t, e_i, i);
         fim
         r ← r_i;
         espera por |conf| > |P|/2 ∨ r_i > r;
         se |conf| > |P|/2 então
            envia Decisao(t, e_i);
         fim
      fim
   fim
fim
```

Figura C.5: Componente carregador para transportar uma estimativa.

```
carregador Suspeita para o hospedeiro Consenso usa
 t,   /* iteração */
 j   /* processo emissor da suspeita */
faz
   espera por decidido_{t-1};
   se ¬decidido_t então
      susp ← susp ∪ {j};
      se |susp| = 1 então
         espera por |susp| > |P|/2 ∨ r_i > r;
         se |susp| > |P|/2 então
            envia Cancelamento(t, i, e_i);
         fim
      fim
   fim
fim
```

Figura C.6: Componente carregador para transportar uma suspeita.

O carregador associado à estimativa (Figura C.5) serve como confirmação que o seu emissor recebeu essa estimativa. Se é a primeira estimativa recebida pelo processo em relação à ronda actual, então:

- se não é o coordenador, retransmite a estimativa de modo a confirmar a sua boa recepção;
- espera pela terminação da ronda ou pela reunião de uma maioria de confirmações;
- no caso de ser reunida uma maioria de confirmações então envia um carregador de decisão para todos os processos.

O componente carregador que transporta uma suspeita (Figura C.6) é semelhante ao carregador da estimativa, na medida em que procura reunir uma maioria, neste caso de suspeitas, relativa à mesma ronda, caso em que emite uma mensagem de cancelamento para todos os processos participantes.

O carregador de cancelamento (Figura C.7) é ainda semelhante aos dois anteriores, na medida em que também procura reunir uma maioria. No entanto, quando isso acontece não emite nenhuma carregador, limitando-se a mudar a ronda do processo local, o que permite receber estimativas dessa ronda ou assumir o papel de coordenador.

O carregador de decisão (Figura C.8) termina uma iteração, assegurando que da primeira vez que é executado em cada processo:

- é adoptada a estimativa final;

```
carregador Cancelamento para o hospedeiro Consenso usa
  t,   /* iteração */
  j,   /* processo emissor do cancelamento */
  e_j  /* estimativa mais recente de j */
faz
   espera por decidido_{t-1};
   se ¬decidido_t então
      canc ← canc ∪ {j};
      se |canc| = 1 então
         espera por |canc| > |P|/2 ∨ r_i > r;
         se |canc| > |P|/2 então
            r_i ← r_i + 1;
         fim
      senão
         e_i ← maior(e_i, e_j);
      fim
   fim
fim
```

Figura C.7: Componente carregador para transportar uma suspeita.

```
carregador Decisão para o hospedeiro Consenso usa
  t,   /* iteração */
  e_j  /* estimativa decidida */
faz
   espera por decidido_{t-1};
   se ¬decidido_t então
      e_i ← e_j;
      decidido_t ← V;
      reenvia;
   fim
fim
```

Figura C.8: Componente carregador para transportar uma decisão.

- a iteração é assinalada como terminada;
- o carregador de decisão é retransmitido.