UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
Departamento de Informática

Plataforma de suporte ao desenvolvimento e composição de malhas de protocolos

Hugo Alexandre Tavares Miranda
(Licenciado)

Dissertação para obtenção do grau de
Mestre em Informática

Março de 2001

Plataforma de suporte ao desenvolvimento e composição de malhas de protocolos

Hugo Alexandre Tavares Miranda

Dissertação submetida para provas

de mestrado em

Informática

Departamento de Informática

Faculdade de Ciências da Universidade de Lisboa

Lisboa

Março de 2001

Dissertação realizada sob a orientação do

Doutor Luís Eduardo Teixeira Rodrigues

Professor Auxiliar
Faculdade de Ciências da Universidade de Lisboa

Este trabalho foi parcialmente suportado pelo projecto PRAXIS/C/EEI/12202/1998,
TOPCOM

Resumo

Muitas das aplicações actuais requerem a utilização simultânea de diversos canais de comunicação, com diferentes requisitos de Qualidades de Serviço (QdS). Para que o desempenho obtido seja óptimo, o programador deverá poder especificar uma composição de protocolos que satisfaça exactamente os requisitos da sua aplicação.

A dissertação apresenta exemplos de aplicações cujos requisitos só são cabalmente satisfeitos por composições de protocolos que exibem inter-dependências complexas entre os vários canais de comunicação. Mostra que as arquitecturas de comunicação existentes dificultam a construção destas estruturas complexas, relegando a sua concretização para as aplicações. A dissertação investiga arquitecturas alternativas que evitam estas limitações. A arquitectura proposta suporta diferentes tipos de composição inter-protocolo e intra-protocolo (ou seja, entre diferentes sessões de um mesmo protocolo). Esta arquitectura é validada através do desenvolvimento de um protótipo e da análise de desempenho de diferentes malhas que ilustram os diversos tipos de composição.

Abstract

Many actual applications require the simultaneous use of several communication channels with different Quality of Service requirements. To optimize performance, the programmer should be allowed to specify a protocol composition satisfying exactly the application requirements.

This thesis presents several examples of applications with requirements that are only completely satisfied by protocol compositions expressing complex inter-channel dependencies. It shows that actual communication architectures raise difficulties to the construction of those complex structures, putting the burden on the application programmer. The thesis researches for alternative models, avoiding these limitations. The proposed architecture supports different composition types, allowing to express inter and intra channel constraints. The model is validated by a prototype and by the performance analysis of some compositions.

Palavras Chave

Sistemas distribuídos,
Composição de protocolos,
Qualidade de serviço

Keywords

Distributed systems,
Protocol composition,
Quality of Service

Agradecimentos

A conclusão deste trabalho marca um momento de viragem no meu percurso, pessoal e profissional. O conjunto de pessoas que marcam este ponto de viragem é em parte coincidente nas duas vertentes. Esta dissertação não teria sido possível sem a sua colaboração. Aproveito por isso para expressar aqui os meus agradecimentos aqueles que mais significativamente deram a sua contribuição.

Os meus pais e as minhas irmãs foram fundamentais na vertente pessoal. O incentivo que, cada um de sua forma, tão bem me soube transmitir e a disponibilidade que sempre manifestaram foram cruciais para que atingisse este objectivo.

À Cristina devo uma parte significativa destes resultados. A sua paciência tem sido insuperável: nos fins de semana e noitadas passados ao computador, na repartição injusta de tarefas, ao ouvir com interesse explicações sobre assuntos para ela irrelevantes. A motivação que sempre me transmitiu foi essencial para que este trabalho tivesse uma conclusão.

O Departamento de Informática da Faculdade de Ciências e o Grupo Navigators, onde englobo todos os seus membros, tiveram uma importância continuada no trabalho que apresento. Ao proporcionar a oportunidade para que desse os meus primeiros passos na investigação através de um programa de estágios com mais prejuízo que benefícios para os membros e ao suportar económica e cientificamente a investigação. As sugestões e críticas apresentadas e a paciência na revisão dos artigos entretanto submetidos, sobretudo por parte do Engº António Casimiro, foram importantíssimos no desenrolar deste trabalho. Não deixou também de ser importante a forma como acarinharam o resultado final, pela sua divulgação e pelo empenho que colocaram para que o *Appia* fizesse parte dos projectos em que participam.

De entre os membros do Grupo Navigators, dois se destacam por fazerem parte da equipa que mais activamente colaborou no desenvolvimento do *Appia*. O Alexandre Pinto que para além do desenvolvimento do código, fez sempre questão em contribuir também para os aspectos arquitecturais do sistema. Os confrontos entre a sua visão prática e o conhecimento que recolhi do trabalho relacionado foram fundamentais para que se atingisse o ponto de equilíbrio.

O Professor Luís Rodrigues ultrapassou largamente a sua missão de orientador. Foi quem me incentivou a participar no programa de estágios tendo apostado num ainda aluno de licenciatura, delegando-lhe o trabalho de investigação que me despertou o interesse por esta actividade. Durante os dois anos e meio que durou o curso de mestrado, foi um orientador incansável, não só na vertente cientifica mas também na vertente pessoal. A sua disponibilidade e partilha de tempo, recursos e conhecimento reflectem a sua orientação e acompanhamento constante e empenhado neste projecto, sem os quais este trabalho nunca teria existido. Também ao nível pessoal: a motivação que me soube transmitir ao longo deste percurso foi crucial. Só com a insistência que manteve na manutenção de um bom ritmo de trabalho, que se reflectiu nas publicações realizadas, foi possível apresentar agora esta dissertação.

Lisboa, Março de 2001

Hugo Alexandre Miranda

À Cristina e ao Gonçalo

Índice

Lista de Figuras

Lista de Tabelas

1 Introdução

> Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway.
>
> Andrew S. Tanenbaum, *Computer Networks*

Nos últimos anos tem-se assistido à crescente valorização da capacidade de intercomunicação dos sistemas informáticos. Actualmente, quase todos os equipamentos são comercializados com dispositivos que permitem o acesso à Internet.

A tecnologia tem acompanhado os requisitos da sociedade: evoluções ao nível físico da comunicação –responsável pela propagação do sinal entre os pontos comunicantes– têm contribuído para a diminuição dos custos da comunicação, redução das taxas de erro e aumento da velocidade de transferência.

Cabe agora aos programadores explorar de forma adequada os recursos que lhe são oferecidos. O primeiro passo nesse sentido foi tomado com o advento da Internet: um conjunto de protocolos que cooperativamente oferecem serviços básicos a uma rede universal de computadores.

1.1 Composição de protocolos

A modularidade dos protocolos é um conceito que tem vindo a afirmar-se e é actualmente quase universal. Segundo ele, o serviço de rede deve ser decomposto em pequenas unidades lógicas, cada uma delas valorizando a comunicação com um conjunto limitado de propriedades.

Alguns dos modelos de referência, nomeadamente, o *Open Systems Interconnected* (OSI) da *International Standards Organization* (ISO) (ISO, 1982) e o TCP/IP assentam neste pressuposto e representam os seus conjuntos de propriedades como pilhas de protocolos.

Um dos temas que recorrentemente tem sido objecto de investigação é a procura do ambiente ideal para a execução e composição de protocolos. As universidades de Cornell e do Arizona apresentaram nos últimos anos diferentes abordagens ao problema. A primeira com os sistemas Isis, Horus e Ensemble e a segunda com o x-Kernel, o Cactus e o Coyote. Para além destes grupos, que mantêm uma actividade sistemática na área, não é difícil encontrar outras abordagens ao assunto. É o caso do Bast, desenvolvido na École Polytechnique Fédérale de Lausanne (EPFL) e do Groupz, da Universidade do Minho. A insuficiência dos ambientes de composição genéricos tem servido como motivação para que grupos de outras áreas, sobretudo da multimédia e do trabalho cooperativo, apresentem as suas próprias propostas. Este é o caso do Collaborative Computing Transport Layer (CCTL), desenvolvido na North Carolina State University. O Grupo Navigators de Investigação em Sistemas Distribuídos tem também experiência na área, acumulada com o desenvolvimento do sistema xAmp (eXtended Atomic Multicast Protocol), que disponibiliza às aplicações um conjunto modular de protocolos de comunicação em grupo.

A tarefa tem sido alvo de diversas aproximações que cruzam factores tão diversificados como a gestão de memória, o momento da composição dos protocolos (em tempo de compilação ou de execução) e os métodos de comunicação entre eles. O tema está longe da exaustão e são poucos os consensos obtidos.

Esta dissertação contribui para a área propondo um novo sistema, denominado *Appia*. Na sua definição foram absorvidos conceitos de grande parte dos outros sistemas apresentados. No entanto, o sistema alarga os horizontes dos modelos anteriores ao proporcionar uma flexibilidade acrescida, que satisfaz os requisitos que emergiram com a evolução das redes de computadores. O resultado é um sistema flexível, codificado numa linguagem cuja utilização regista uma expansão considerável. O *Appia* utiliza os pontos mais fortes de cada um dos sistemas apresentados para introduzir as suas próprias inovações.

O sistema *Appia* tem vindo, desde a sua distribuição, a ser utilizado em vários projectos de investigação nacionais e internacionais. Estes projectos cobrem um largo espectro de aplicações: segurança, sistemas de informação, ambientes virtuais multi-utilizador e aplicação de informação semântica à difusão fiável de mensagens.

1.2 Estrutura da dissertação

A dissertação é composta por 6 capítulos, estruturados da forma que a seguir se apresenta.

O capítulo 2 apresenta um conjunto de conceitos relacionados com os sistemas de suporte à comunicação. A partir da definição elementar de protocolo, o capítulo prolonga-se pelos factores que condicionam o desenho destes sistemas. Para cada um, são enumeradas as diferentes opções disponíveis e analisadas as suas consequências.

O capítulo 3 apresenta uma panorâmica sobre o estado-da-arte actual, analisando os sistemas x-Kernel, Coyote, Horus, Ensemble, Bast, Conduit+ e Groupz. O ênfase é colocado no modelo de composição apresentado por cada um. São também abordadas outras características, particularmente aquelas que se relacionam com os temas discutidos no capítulo 2. As características do conjunto dos sistemas são comparadas no final do capítulo.

O início do capítulo 4 apresenta dois exemplos de requisitos de comunicação complexos. Para cada um deles são discutidas possíveis soluções, utilizando o trabalho relacionado. Estes dois exemplos complementam-se na estruturação de um modelo de composição alternativo, que não pode ser conseguido em nenhum dos sistemas apresentados. O capítulo continua com uma descrição do *Appia*: um sistema que suporta composições complexas de protocolos. O desenho e algumas características da concretização do *Appia* concluem o capítulo.

O capítulo 5 avalia o *Appia* em diversas vertentes. Começa por demonstrar a capacidade do sistema em satisfazer os requisitos das aplicações utilizadas como motivação. Esta secção é também utilizada como pretexto para apresentar exemplos de algoritmos e código para a especificação de composições e concretização de protocolos. O capítulo termina com uma breve avaliação do desempenho sob três perspectivas. As duas primeiras analisam o impacto de facilidades específicas do sistema. A última apresenta uma visão crítica sobre o desempenho global do *Appia*, apontando algumas justificações para os resultados obtidos.

O capítulo 6 conclui este documento e lança algumas ideias para a continuação do trabalho.

2

Conceitos

Os ambientes de suporte à composição de micro-protocolos são sistemas complexos. No desenho do sistema é necessário ter em consideração um conjunto lato de factores estruturais, como o momento em que a ligação das camadas será realizada, ou o enquadramento que a aplicação terá com o sistema operativo. Para cada um deles, os arquitectos do sistema terão que optar por uma das aproximações disponíveis. Não existindo a "solução ideal", impõe-se conhecer, *à priori*, a totalidade das opções disponíveis e o impacto de cada uma delas no resultado final.

Este capítulo, apresenta uma panorâmica sobre os factores que condicionam os ambientes de suporte a protocolos de redes de computadores. Para cada um desses factores são apresentadas as diferentes opções, e analisadas as suas vantagens e desvantagens.

2.1 Protocolo

Protocolo é a especificação de uma abstracção de comunicação através da qual um conjunto de participantes troca um conjunto de mensagens (Hutchinson & Peterson, 1991). Um protocolo tem duas classes de funções (Clark & Tennenhouse, 1990):

Manipulação de dados. Que realizam operações de leitura e alteração dos dados transferidos e que por sua vez são subdivididas em seis categorias: transferência de, e para a rede; detecção de erros; armazenamento para possível retransmissão; cifra; transferência de, e para o espaço de endereçamento da aplicação e formatação ao nível de apresentação;

Controlo da transferência de dados. As operações mais comuns dedicadas ao controlo da comunicação dos dados são: controlo de congestão e de fluxo; detecção de problemas de transmissão; confirmações; multiplexagem e delimitação.

Formalmente, um protocolo é uma função P que, a partir de um estado S (o conjunto de variáveis que o protocolo mantém) e de um evento E, gera um estado S' e um conjunto (possivelmente vazio) de eventos, ou seja (van Renesse *et al.*, 1997),

$$P(S, E) \rightarrow (S', \{E_1, E_2, \ldots\}) \tag{2.1}$$

Os protocolos tornam-se tendencialmente complexos porque são concebidos para satisfazer os requisitos de diferentes aplicações e porque as actualizações são geralmente realizadas de forma a que mantenham a compatibilidade com as versões anteriores (O'Malley & Peterson, 1992).

A operação de composição de protocolos é definida pela aplicação do conjunto de eventos resultante da aplicação de E ao primeiro protocolo no seguinte. O estado da composição de protocolos é a união dos estados individuais (van Renesse *et al.*, 1997). Este princípio é frequentemente utilizado em diversas outras áreas da informática, onde é globalmente denominado de **modularidade**.

A aproximação conceptual de *dividir para conquistar* foi adoptada pelos dois modelos de referência mais comuns: o modelo Open Systems Interconnected (OSI) (ISO, 1982) e o modelo TCP/IP (Postel, 1981a; Postel, 1981b). Em ambos, as propriedades apresentadas à aplicação resultam da composição ordenada de um conjunto de protocolos. A decomposição conceptual do modelo TCP/IP está representada na figura 2.1. Com esta abordagem, o serviço prestado por cada um dos protocolos pode ser analisado de forma independente dos restantes. Por outro lado, a existência de níveis abstractos, onde são especificadas as propriedades a fornecer e não os algoritmos, torna possível concretizar diversos protocolos que satisfaçam as propriedades requeridas mas com características de desempenho diferentes. Por fim, a análise global do serviço prestado pela composição torna-se mais clara.

Podem ser encontrados quatro tipos de relações entre protocolos (Bhatti *et al.*, 1998):

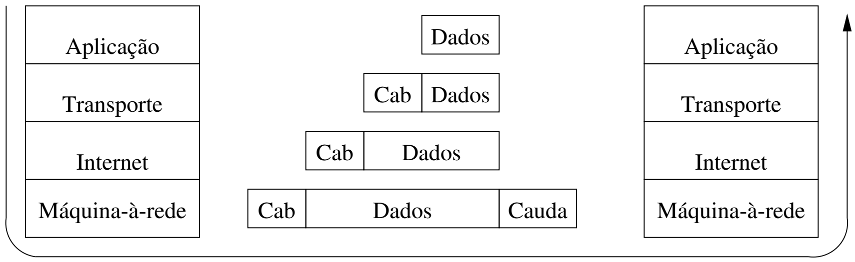

Figura 2.1: Modelo de referência TCP/IP (Tanenbaum, 1996).

Conflito Dois protocolos estão em conflito se não podem ser compostos, seja devido à impossibilidade de garantir as suas propriedades simultaneamente ou a opções tomadas durante a sua concretização.

Independência Os protocolos m_1 e m_2 são independentes se $m_i, i \in \{1, 2\}$ pode ou não integrar uma composição onde $m_j, j \in \{1, 2\} \wedge i \neq j$ participa. Em qualquer das combinações, os protocolos terão que satisfazer as propriedades especificadas.

Dependência O protocolo m_1 depende do protocolo m_2 se m_2 tiver que estar presente na composição, para que as propriedades de m_1 sejam asseguradas.

Inclusão Um protocolo m_1 inclui o protocolo m_2 se m_1 assegura uma propriedade estritamente mais forte que m_2, sem que exista uma relação de dependência entre eles. Neste caso, pode-se concluir que a composição de m_1 com m_2 é redundante.

Uma das vantagens da modularidade de protocolos é a possibilidade da sua reutilização em diferentes composições.

Existem dois factores que condicionam a aplicação de protocolos a plataformas e serviços para os quais não foram inicialmente concebidos. **Reusabilidade** é definida como a possibilidade de utilização de um protocolo em múltiplos serviços, independentes entre si e **configurabilidade** como o grau com que um protocolo pode ser combinado com outros, para proporcionar variantes úteis de um dado serviço (Bhatti *et al.*, 1998).

A reusabilidade é uma propriedade intuitiva que se demonstra de difícil concretização por diversas razões. Em primeiro lugar, a concretização dos protocolos depende de uma plataforma que impõe, de forma mais ou menos forçada, um conjunto de restrições e facilidades. A adaptação de um protocolo a uma outra plataforma pode não ser possível devido às diferentes facilidades oferecidas por cada uma, como por exemplo, o conteúdo das estruturas de dados partilhadas entre protocolos. Um outro problema são as dependências, não explícitas mas de programação, que surgem entre os protocolos. Por exemplo a projecção entre um evento e o seu nome, que deverá ser conhecida de todos os protocolos interessados. Para o caso dos eventos definidos pelo programador, esta correspondência é estabelecida arbitrariamente e dificilmente possibilitará a reutilização dos protocolos noutras configurações.

A configurabilidade depende da inexistência de conflitos e dependências no acesso aos atributos das mensagens e às estruturas de dados partilhadas (Bhatti *et al.*, 1998).

Um protocolo é **polimórfico** se aceitar, de forma transparente, diversos tipos de dados. Uma aplicação evidente deste conceito é a dos endereços. Embora a informação sobre endereços tenha que circular por diversos protocolos, ela não terá interesse para um número significativo deles. Protocolos polimórficos deverão aceitar qualquer tipo de endereço de outros protocolos, alargando assim a sua reusabilidade (O'Malley & Peterson, 1992).

2.2 Composição de protocolos

O número ideal de protocolos a compor tem sido assunto de discussão. O modelo OSI propõe sete enquanto que o modelo TCP/IP apresenta quatro. Recentemente tem-se assumido que o número de protocolos não pode ser determinado à partida. Alguns autores defendem que este número deve estar dependente da mensagem (O'Malley & Peterson, 1992), enquanto outros condicionam-no às propriedades pretendidas mantendo-o fixo independentemente da mensagem (van Renesse *et al.*, 1996).

Na composição de protocolos a comunicação é indirecta. Ou seja, entidades correspondentes comunicam entre si passando mensagens a um protocolo de que dependem. A informação de controlo, necessária para que os protocolos forneçam as propriedades que lhe são requeridas é adicionada aos dados provenientes do protocolo de nível superior sob a forma de cabeçalhos ou caudas que são extraídos pelo seu par no destinatário.

Uma composição de protocolos define-se como **útil** se satisfaz as necessidades de comunicação da aplicação sobre uma determinada rede. Desejavelmente, as composições devem também ser **minimais**. Ou seja, não oferecer propriedades que a aplicação não deseja e que introduzem custos adicionais de comunicação (van Renesse *et al.*, 1997).

2.2.1 Grafos de protocolos

As composições de protocolos podem ser representadas por grafos acíclicos dirigidos, neste caso denominados grafos de protocolos. Os vértices do grafo representam os protocolos e as arestas relações de dependência. Isto é, se o protocolo A envia mensagens para as entidades correspondentes utilizando as propriedades do protocolo B então existe uma aresta terminada em B com início em A (O'Malley & Peterson, 1992).

Existem duas formas usuais de composição de protocolos: em árvore e em pilha.

2.2.1.1 Composição em árvore

Uma **árvore** é um grafo conexo e acíclico. Na composição em árvore, a raiz é o protocolo mais próximo da rede, enquanto as folhas representam as aplicações. A figura 2.2 apresenta a composição em árvore dos protocolos do modelo de referência TCP/IP.

Em alguns casos, a representação em árvore mostra-se demasiado simplista, obtendo-se na composição alguns ramos que não respeitam a definição. Tomando ainda a figura 2.2, a composição não representaria uma árvore se uma das aplicações utilizasse simultaneamente os protocolos TCP e UDP, ou se o protocolo ARP (Plummer, 1982) fosse incluído na representação.

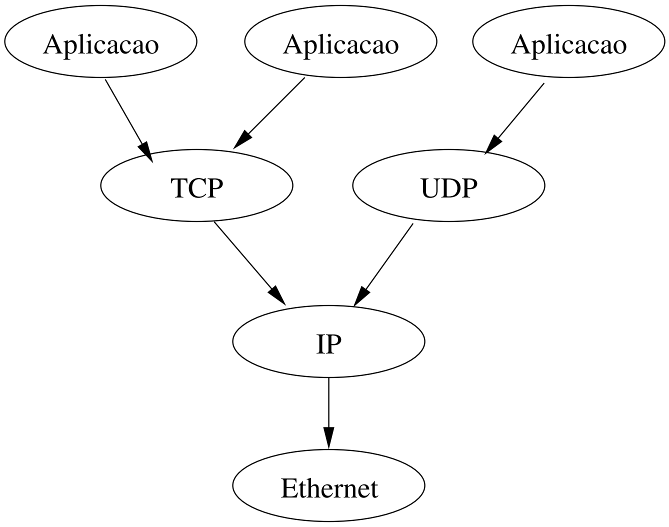

Figura 2.2: Representação em árvore dos protocolos do modelo de referência TCP/IP

2.2.1.2 Composição em pilha

Os protocolos podem também ser associados sob a forma de pilhas. onde o protocolo que concretiza o nível físico é a camada inferior e a aplicação que utiliza o serviço a camada superior. Devido à eliminação das arestas, as hipóteses de dependência entre protocolos tornam-se mais fracas: poderá ou não existir uma dependência de um protocolo para um outro, disposto na pilha abaixo deste.

Este modelo corresponde à compressão dos grafos de protocolos de forma a que todos os vértices, com excepção dos extremos, sejam origem e destino de um único arco. As pilhas de protocolos impõem maiores restrições à composição. As implicações de cada um dos modelos serão discutidas mais à frente neste texto.

Um exemplo de composição em pilha é o disponibilizado pelos filtros do ambiente UNIX. Todos os programas que obtenham dados do descritor de ficheiros denominado *stdin* e coloquem os resultados no descritor denominado *stdout* podem ser compostos linearmente através de operadores especiais de composição (O'Malley & Peterson, 1992). A unidireccionalidade é uma característica particular deste modelo. Cada um dos programas compostos comporta-se como consumidor da informação gerada por apenas um dos programas e como produtor de informação para um outro.

2.2.1.3 Composições mistas

A composição em árvore favorece a congestão nos protocolos de nível inferior, resultante da concentração das mensagens. A solução mais eficiente será aquela que adia a congestão para o ponto em que se torna inevitável: o acesso à rede, tipicamente representado pela raiz da árvore de protocolos (Tennenhouse, 1989).

Os grafos resultantes desta solução são uma mistura da composição em árvore com a composição em pilha: da raiz da composição surge um conjunto de pilhas. Esta solução foi adoptada pela plataforma de comunicação em grupo Ensemble (Hayden, 1998). Uma particularidade do Ensemble é a realização do encaminhamento das mensagens pelo núcleo de suporte, ou seja, é este que desempenha o papel de raiz da árvore de composição. Para o programador, cada pilha é totalmente independente das restantes. O Ensemble será analisado em mais pormenor na secção 3.5.

2.3 Plataformas de composição

Uma plataforma de composição de protocolos é um conjunto de ferramentas que facilita a integração de protocolos, codificados como módulos independentes. A estrutura e serviços que estas plataformas devem apresentar tem sido alvo de estudo de diversos grupos de investigação e é ainda um assunto em discussão. Nas páginas seguintes são apresentadas algumas das opções que distinguem os vários modelos propostos.

2.3.1 Plataformas monolíticas e hierárquicas

Independentemente da granularidade da decomposição dos protocolos, as plataformas de composição podem ser divididas em duas classes:

Monolíticas As plataformas monolíticas caracterizam-se por terem um conjunto imutável de protocolos embebido no seu código. Em alguns casos, o conjunto de protocolos utilizados pela aplicação pode variar. O Isis (Birman & van Renesse, 1994) é um exemplo de plataformas monolíticas.

Hierárquicas As plataformas hierárquicas têm um conjunto dinâmico de protocolos, opcionalmente incluídos no código final a utilizar pela aplicação.

As plataformas hierárquicas podem ainda ser caracterizadas pelo momento em que o código dos protocolos é definitivamente associado (ligado) ao código da plataforma. É possível encontrar plataformas que requerem a especificação dos protocolos em tempo de compilação e outras que toleram a alteração dos protocolos durante a execução. Estas últimas são usualmente referidas como suportando **ligação tardia**.

São exemplos de plataformas hierárquicas o Horus (van Renesse *et al.*, 1996), o Coyote (Bhatti *et al.*, 1998) e os Local Supporting Environments (LSE) (Fonseca, 1994).

O conhecimento que as arquitecturas monolíticas têm da estrutura do grafo em que estão inseridas pode ser utilizado em diversas optimizações (O'Malley & Peterson, 1992; Hayden, 1998): menor consumo de recursos computacionais, partilhando informação entre protocolos, e cabeçalhos de menor dimensão, uma vez que atributos comuns a vários protocolos não necessitam de ser repetidos.

As optimizações nas plataformas monolíticas são conseguidas sobretudo através do aumento da dependência das concretizações dos protocolos. As plataformas hierárquicas, por sua vez, contrapõem uma maior flexibilidade, que compensa, noutras vertentes, as optimizações dos sistemas monolíticos (O'Malley & Peterson, 1992; van Renesse *et al.*, 1995a; Hayden, 1998):

- Uma vez que os protocolos concretizam funções simples, incentivam a reusabilidade e portanto contribuem para a simplicidade de novos sistemas de comunicação. O desenvolvimento de protocolos é mais rápido e de mais fácil depuração do que concretizações monolíticas análogas;
- Numa plataforma dinâmica, os programadores de aplicações podem seleccionar

apenas as propriedades necessárias. Os custos que a aplicação suporta correspondem aos recursos que usa e não aos que a plataforma proporciona;

- Diferentes concretizações do mesmo protocolo podem fazer variar a sua eficiência em condições ambientais distintas.[1] Num sistema dinâmico o programador pode seleccionar a variante que melhor se adapta às condições em que a aplicação será executada;

- Uma arquitectura dinâmica adapta-se mais facilmente às mudanças das tecnologias de rede;

As arquitecturas em camadas são penalizadas pela independência dos protocolos. Para manter a modularidade, as plataformas têm que oferecer mecanismos de controlo de concorrência, de gestão de filas de eventos e de suporte à ligação dos protocolos. A partilha de informação de estado é reduzida ou mesmo eliminada, gerando redundância que se propaga para os cabeçalhos adicionados às mensagens.

2.3.2 Concorrência

Nas plataformas de suporte à distribuição podem existir diversos eventos a ser processados simultaneamente. Parece por isso vantajoso a utilização de múltiplos fluxos de controlo. De entre as entidades participantes, os fluxos de controlo podem ser atribuídos a mensagens ou a protocolos. Uma terceira solução não utiliza fluxos de controlo concorrentes. Neste caso, um único fluxo de controlo é responsável pelo processamento dos eventos. Este modelo pode ser flexibilizado, permitindo a utilização de fluxos auxiliares para simplificação das operações de suporte, por exemplo na gestão de temporizadores.

A atribuição de processos aos protocolos sofre de problemas de desempenho graves, principalmente associados ao tempo necessário para a realização de mudanças de contexto (Hutchinson & Peterson, 1991; Silberchatz *et al.*, 2000). Embora atenuado, o problema persiste quando são utilizadas actividades.

[1] Por exemplo, o desempenho dos diferentes algoritmos que concretizam protocolos de ordenação total difere substancialmente com as condições da rede que estão a utilizar.

A utilização de um fluxo de controlo por cada evento no grafo, dificulta a aplicação da política *primeiro-a-entrar-primeiro-a-sair* aos eventos, no seu percurso pelo grafo. Este requisito é desejável na maioria das situações, nomeadamente, sempre que se criem relações de causa-efeito entre eventos. Para contornar a arbitrariedade do escalonador de processos, é necessário recorrer a artifícios de controlo de concorrência. A concretização do TCP (Postel, 1981b) no x-Kernel, por exemplo, define uma zona crítica que inclui todos os protocolos acima, para evitar a entrada de mais de uma mensagem nessa zona (Hutchinson & Peterson, 1991).

A utilização de diversos fluxos de controlo apresenta ainda outras desvantagens. O sistema tem o seu desempenho prejudicado pelo custo associado à utilização das primitivas de controlo de concorrência. Por outro lado, a programação torna-se mais complexa, aumentando o tempo de codificação e depuração (Birman, 1996). Finalmente, não é claro que a utilização de actividades em sistemas operativos que não os suportem explicitamente aumente o desempenho das aplicações. Uma concretização da plataforma TIMEWHELL (Mishra & Yang, 1998), utilizando um único fluxo de controlo, veio a mostrar-se substancialmente mais eficiente que uma outra equivalente, utilizando actividades.

2.3.3 Encaminhamento das mensagens na composição

Nas plataformas que suportam composição em árvore é necessário assegurar a entrega das mensagens ao conjunto correcto de protocolos. Pela sua natureza, as composições em pilha não necessitam de informação de encaminhamento das mensagens.

O encaminhamento pode ser realizado pelos protocolos ou pelo núcleo de suporte à composição. Quando o encaminhamento é realizado pelos protocolos, o seu cabeçalho deve conter informação que lhe permita determinar o próximo destinatário da mensagem. É essa a função do campo *protocol* do cabeçalho do protocolo IP e dos números de porto dos protocolos UDP e TCP. Cada protocolo compara a informação de encaminhamento que recebe no cabeçalho com tabelas internas e determina a próxima camada a que entregará a mensagem. Se o encaminhamento for realizado pelo sistema de suporte à composição, cabe-lhe a responsabilidade de adicionar à mensagem um identificador que permita, ao receptor, determinar o conjunto de protocolos que a mensagem deve visitar. Em alternativa, o identificador da sequência de protocolos pode ser trocado entre as camadas mais baixas da composição. Aquando da recepção, essa camada converte-o num identificador interno, conhecido pelo núcleo de suporte à composição.

A opção de encaminhamento de mensagens tem impacto na capacidade de intervenção do núcleo na execução dos protocolos. O encaminhamento realizado pelas camadas retira ao núcleo a possibilidade de intervir na entrega das mensagens. Cada camada é responsável por invocar o procedimento de recepção de mensagem do protocolo seguinte. A concretização pelo núcleo de algumas facilidades, como a ligação tardia dos protocolos, torna-se bastante mais complexa. Por outro lado, ocorre um consumo evitável de recursos nas projecções que cada camada efectua às tabelas de encaminhamento de mensagens. Por outro lado, quando o encaminhamento é realizado pelo núcleo do sistema de suporte ocorre necessariamente um aumento da sua complexidade. A mediação do núcleo na propagação das mensagens implica também a invocação de mais procedimentos e de uma mais complexa coordenação dos processos.

2.3.4 Enquadramento no sistema

Os protocolos de comunicação podem ser incluídos no sistema de três formas (Clark, 1982): no núcleo do sistema operativo, como um processo ou num processador dedicado. Cada uma das aproximações apresenta vantagens e desvantagens.

A instalação dos protocolos num processo apresenta uma maior independência ao sistema operativo. O programador do protocolo não necessita de conhecer os pormenores da concretização do sistema operativo e este, por sua vez, fica protegido de eventuais erros que levassem à sua instabilidade. O aumento de desempenho obtido com a inclusão dos protocolos no sistema operativo é comprometido pela redução da portabilidade da plataforma. O sistema operativo fica também limitado nas possibilidades de escalonamento. Todo o sistema fica por isso mais vulnerável se, por exemplo, forem recebidas grandes quantidades de mensagens.

A opção de concretização dos protocolos num processador separado oferece claramente o melhor desempenho: o ambiente de execução pode ser concebido para satisfazer especificamente as particularidades dos protocolos de comunicação em redes de computadores. No entanto, esta aproximação pode ser vista como um adiamento do problema, uma vez que será necessário definir regras adequadas para a comunicação entre os dois processadores. A adição de um processador dedicado apresenta também penalizações ao custo final do sistema.

Independentemente da opção escolhida, um problema incontornável é o da entrega e recepção das mensagens às aplicações, executadas em modo utilizador. Se os protocolos forem também executados em processos ou embebidos na aplicação cliente, o problema colocar-se-á na transferência dos dados entre os protocolos e o dispositivo. Se, pelo contrário, os protocolos forem integrados no núcleo, o atraso registar-se-á na passagem dos dados da aplicação para os protocolos. A solução encontrada pelo x-Kernel é descrita na secção 3.2.

2.3.5 Desempenho

As plataformas de suporte à comunicação, como prestadoras de um serviço às aplicações, têm requisitos de desempenho muito elevados. Esta tarefa é dificultada por competirem, com a aplicação, pelos recursos do sistema em que ambas são executadas.

Ao nível da concretização podem ser encontrados diversos factores, alguns deles já referidos anteriormente, que condicionam o desempenho (Hutchinson & Peterson, 1991; van Renesse *et al.*, 1995a; Clark, 1982; Clark & Tennenhouse, 1990) das plataformas de suporte à composição. Nos factores externos à composição encontram-se por exemplo, as limitações de *hardware*, o desempenho dos sistemas em que são executados e as políticas de escalonamento dos sistemas operativos. Por sua vez, o núcleo de suporte à composição é responsável pela política de atribuição de processos e pelas ferramentas que disponibiliza para a gestão de tampões. Por fim, o desempenho é também influenciado pelos protocolos que constituem a composição e pela própria aplicação. Neste caso, as responsabilidades são a dimensão dos dados da aplicação e dos cabeçalhos do protocolo, com impacto no tempo de transmissão e no tempo de processamento por alguns protocolos; o algoritmo de controlo de fluxo; o tratamento dos cabeçalhos; as formatações do nível de apresentação e a generalização dos protocolos.

2.3.5.1 Delimitação de nível de aplicação

Uma unidade de dados aplicacionais[2] (ADU) é o menor conjunto de dados que a aplicação pode processar de forma independente (Clark & Tennenhouse, 1990).

Idealmente, as operações de manipulação de dados seriam realizadas sobre ADUs e não sobre as Unidades de Dados Protocolares (PDUs). Esta medida teria um impacto positivo no desempenho global do sistema por três razões: 1) continuidade do processamento pela aplicação e pelo nível de apresentação (que realiza as operações de formatação de dados), independentemente da ocorrência de erros que resultem na perda de ADUs, 2) redução do número de operações de cópia de dados entre o espaço de endereçamento da plataforma e o espaço de endereçamento do processo, 3) em função dos dados recebidos e da sua semântica, a aplicação poderia optar por não solicitar a retransmissão de ADUs não recebidas.

Com as características sugeridas, a delimitação de nível de aplicação delega nos programadores das aplicações as tarefas de detecção e recuperação de mensagens. Tipicamente, estas funções são realizadas por protocolos. A sua ausência na composição impossibilitaria a concretização, a esse nível, de um conjunto de outras propriedades que requerem dos níveis inferiores a fiabilidade de entrega, como por exemplo, a ordenação total das mensagens entre um conjunto de participantes. Surge por isso um efeito de cascata que colocaria sucessivamente mais funções na aplicação e contrariaria os objectivos iniciais das plataformas de suporte à composição: o programador de aplicações teria que se especializar na programação de protocolos. Em alternativa, a plataforma deve oferecer ao programador ferramentas que lhe permitam expressar, para cada mensagem, as propriedades que pretende ver aplicadas.

[2] Do inglês *Application Data Unit*.

2.3.5.2 Generalização dos protocolos

Para assegurarem as possibilidades de reutilização, os protocolos são concebidos por forma a satisfazerem os requisitos do maior número de aplicações possível. Esta generalização provoca problemas de desempenho, uma vez que todos os algoritmos são concebidos para "o caso típico". Optimizações, em alguns casos evidentes, vantajosas apenas para aplicações com determinadas características nunca chegam a ser aplicadas.

O atraso introduzido pelo protocolo TCP antes do envio de uma confirmação de recepção de um segmento, por exemplo, faz todo o sentido em aplicações em que ocorre transporte bidireccional dos dados como no estabelecimento de sesões em sistemas remotos. No entanto, em outro tipo de aplicações, como a transferência de ficheiros, é sabido à partida que esse atraso diminuirá a taxa de transferência de dados por atrasar o envio de outros segmentos e consumirá recursos do sistema do receptor: a adição e tratamento de uma temporização com as consequentes mudanças de contexto associadas.

O conhecimento sobre os algoritmos adequados a cada aplicação é um exclusivo do programador e não pode ser antecipado pelo arquitecto do protocolo. O responsável pelo desempenho do conjunto deve por isso ser o programador. Este problema pode ser resolvido abdicando parcialmente das regras de modularidade dos protocolos decompondo-os em duas vertentes: um "núcleo" do protocolo, contendo os aspectos fundamentais à sua execução, por exemplo, a formatação do cabeçalho; e um conjunto de algoritmos opcionais, que integrariam a aplicação (Clark, 1982). Esta proposta contraria o objectivo de grande parte dos autores de sistemas de suporte à composição: evitar, dos programadores aplicacionais, conhecimentos profundos de protocolos de redes de computadores.

2.3.6 Validação de propriedades

Nos sistemas hierárquicos em que a composição de camadas é arbitrária, é necessário assegurar que determinada combinação mantém o conjunto das propriedades oferecidas individualmente. Este problema advém da possível existência de relações de Conflito e Dependência. Conflito quando combinações de protocolos anulam propriedades oferecidas por outros, dependência quando a um dado protocolo não é possível oferecer a propriedade, por inexistência de um outro no grafo. A detecção destas situações pode ser automatica se o sistema incluir ferramentas de suporte contendo a informação necessária à detecção de potenciais situações de erro.

O Horus (van Renesse *et al.*, 1995a) delega no utilizador a responsabilidade de construir pilhas de protocolos correctas. Para facilitar o processo de validação, existem duas tabelas: uma onde são assinaladas respectivamente as dependências (identificadas como requisitos) de cada camada relativamente a um conjunto de propriedades e a não conflituosidade (sob a forma de herança) e outra que apresenta as propriedades que cada camada satisfaz.

O Ensemble (Hayden, 1998), reforça a distinção entre propriedades e camadas. Em tempo de execução, uma função fornecida com a plataforma, converte as propriedades enunciadas pelo programador num conjunto minimal de camadas com as quais será composta a pilha.

2.4 Sumário

Este capítulo começou por definir um conjunto de conceitos fundamentais para as redes de computadores. Foram cruzadas diferentes definições de *protocolo* na tentativa de estabelecer uma referência comum. As vantagens que apresentam a definição de protocolos com um número reduzido de propriedades foram depois utilizadas para introduzir as diferentes formas de composição.

Em seguida foram abordados os diferentes factores a ter em conta no desenho de um ambiente de suporte à composição. Para cada um, foi enumerado e discutido o conjunto de opções possível.

3

Plataformas hierárquicas

A utilização de sistemas de suporte à composição de protocolos tem vindo a ser alvo de estudo ao longo dos últimos anos. Este capítulo começa pela apresentação de um sistema de composição de referência. Em seguida, são descritas as principais características de alguns desses sistemas. O capítulo é concluido com uma comparação do conjunto que confronta algumas das opções arquitecturais mais importantes.

3.1 Streams

Uma Stream é uma ligação bidireccional entre um processo de utilizador e um dispositivo ou pseudo-dispositivo do sistema operativo Unix. É composto por vários módulos ligados linearmente entre si. Com excepção para algumas variáveis utilizadas no controlo de fluxo, os módulos não partilham memória entre si. Cada módulo disponibiliza apenas um ponto de acesso a cada um dos seus vizinhos, nomeadamente, uma função que aceita mensagens (Ritchie, 1984).

A unidade básica de transferência de informação entre módulos são os blocos. Cada bloco contém um apontador para o início dos dados, um para o fim dos dados e um terceiro para o fim real do bloco, ou seja, para o ponto até onde o fim dos dados pode crescer. O cabeçalho de um bloco especifica o seu tipo, que pode ser de dados ou de informação de controlo. Quando um bloco é processado por um módulo, ele é colocado na fila do módulo seguinte. Cada módulo tem por isso duas filas, contendo os blocos de dados que aguardam processamento em cada sentido.

São definidos dois tipos de informação de controlo: síncrona e assíncrona com os dados. A primeira percorre a composição aguardando processamento na fila de cada módulo. São exemplos de informação de controlo síncrona os delimitadores de mensagens. As mensagens de controlo assíncronas não são atrasadas nas filas de espera e não são por isso ordenadas com os dados. A mensagem que instrui todos os módulos a limparem as suas filas de espera é um exemplo das mensagem de controlo assíncronas.

As Streams são um dos modelos de referência para a composição linear de aplicações e que pode ser extendida para a composição linear de protocolos (van Renesse *et al.*, 1995a). Encontram-se algumas semelhanças entre ambos: os módulos são encapsulados numa interface predefinida, utilizada para a troca de mensagens de dados e controlo. A sua composição é sequencial e pressupõe a troca de mensagens com ambos os módulos adjacentes. Estas características são comuns na maioria das plataformas que serão apresentadas em seguida.

3.2 x-Kernel

O x-Kernel (Hutchinson & Peterson, 1991) é um conjunto de utilitários e bibliotecas para a definição de núcleos de sistema operativo, cada um deles caracterizado pelo grafo de protocolos que utiliza. A plataforma define abstracções para o encapsulamento de protocolos e disponibiliza bibliotecas optimizadas para algumas operações que os protocolos realizam com frequência. As combinações de protocolos são definidas em tempo de compilação do núcleo.

3.2.1 Entidades

O x-Kernel define três objectos de comunicação básicos:

Protocolos são objectos passivos e estáticos que correspondem ao conceito convencional de protocolo.

Sessões são objectos passivos criados dinamicamente. Cada sessão mantém as estruturas de dados e o código necessário ao fornecimento das propriedades de uma instância de um Protocolo.

Mensagens são objectos activos criados dinamicamente, manipulados pelos Protocolos e Sessões. As mensagens contêm dados do utilizador e cabeçalhos de Protocolos.

A figura 3.1 apresenta a relação entre os Protocolos as Sessões e as Mensagens numa composição de parte de um grafo de protocolos TCP/IP sobre Ethernet. Cada protocolo tem associadas sessões que medeiam a ligação com os protocolos imediatamente acima. À medida que atravessam a pilha, as mensagens são encaminhadas para as sessões pelos protocolos, gerando eventuais alterações ao seu estado.

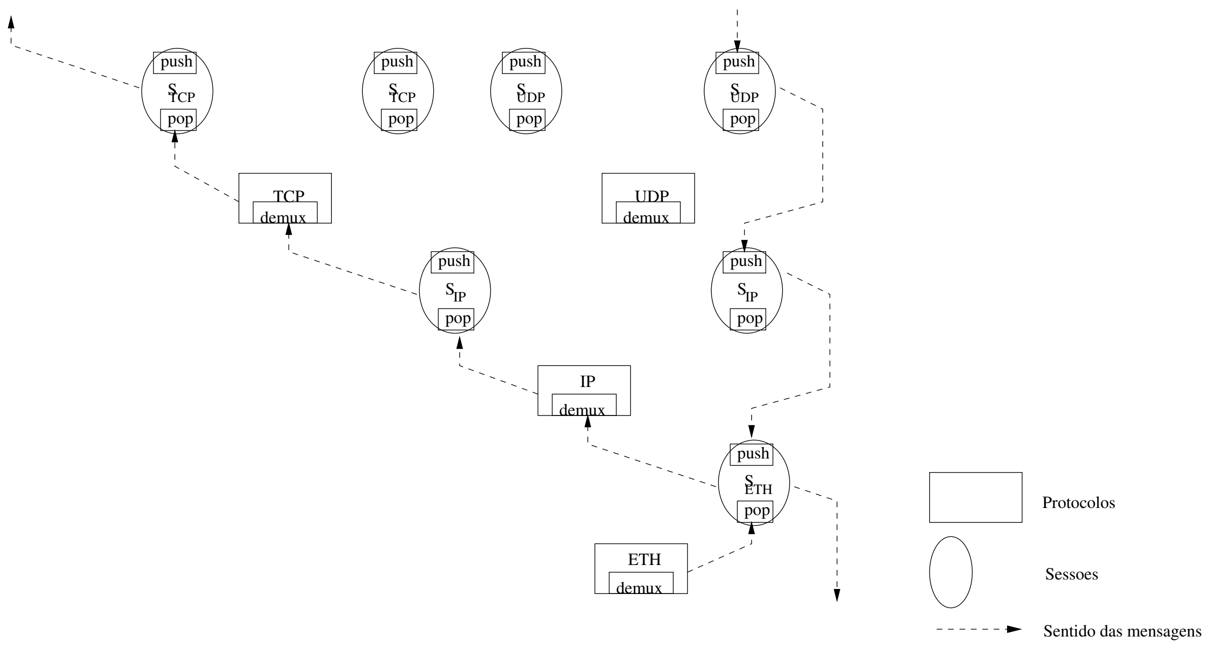

Figura 3.1: Relações entre protocolos, sessões e mensagens do x-Kernel numa pilha TCP/IP (Hutchinson & Peterson, 1991).

3.2.2 Composição de grafos

Cada protocolo é codificado como um módulo independente, que terá necessariamente que satisfazer as especificações da "Interface Uniforme de Protocolos" (UPI) (x-Kernel, 1997). O grafo de protocolos que caracteriza um núcleo é declarado num ficheiro de texto. Em tempo de compilação do núcleo, o ficheiro e os módulos são usados para gerar automaticamente código que crie as ligações entre os protocolos. A ligação é realizada através de chamadas às funções do UPI.

As sessões são criadas em tempo de execução a pedido de qualquer dos protocolos que ligarão. Quando a criação é desencadeada pelo protocolo mais acima na pilha, uma aplicação local sinaliza a intenção de contactar uma aplicação remota.

Antes de uma sessão ser criada pela camada inferior, o nível superior terá que ter comunicado a sua disponibilidade em aceitar dados que lhe sejam enviados com a identificação especificada. Quando a camada inferior receber um pedido nesse sentido, criará a sessão adequada e informará a camada superior da sessão e da identificação do processo remoto (x-Kernel, 1997).

O modelo descrito anteriormente é semelhante ao processo usado no estabelecimento de uma ligação através do protocolo TCP (Postel, 1981b). Neste, a aplicação servidor (protocolo de nível superior) começa por informar o protocolo TCP (de nível inferior) que pretende aceitar ligações solicitadas a um porto. Quando o TCP recebe um pedido de ligação ao porto em causa, notifica o servidor. No x-Kernel, esta notificação corresponde à abertura pelo protocolo TCP de uma sessão, à qual é associado o número do porto. O cliente por sua vez, cria ele próprio uma sessão, usando-a para o estabelecimento da ligação ao porto registado pelo servidor.

3.2.3 Troca de mensagens

O x-Kernel é concretizado sobre um modelo de "actividade por mensagem". Ou seja, a cada mensagem que circula no grafo é atribuída uma actividade, que executará o código necessário ao seu processamento em todos os protocolos e sessões. No modo usual de execução, os protocolos, por fazerem parte do núcleo do sistema operativo, são executados em modo privilegiado e as aplicações são executadas em modo utilizador. A sequência de objectos que uma mensagem visita é dependente do sentido em que viaja.

No sentido descendente, ou seja, no envio, as mensagens visitam apenas as sessões do grafo. Cada sessão adiciona o seu cabeçalho à mensagem e através da invocação da função `push`, entrega a mensagem à sessão seguinte.

Pelo contrário, a sequência de sessões a percorrer pelas mensagens no sentido ascendente tem de ser determinada individualmente por cada protocolo. As mensagens atravessam por isso sequências alternadas de protocolos e sessões. Os primeiros com a função de determinar o encaminhamento correcto nas segundas. Os protocolos recebem as mensagens na função `demux`, que determina qual a sessão correcta para a mensagem e entrega-a pela função `pop`. Cabe à função `demux` a criação de sessões quando tal lhes tenha sido solicitado pelo protocolo de nível superior. No sentido ascendente de uma pilha TCP/IP, por exemplo, (figura 3.1) cabe ao protocolo IP decidir para que protocolo de nível de transporte deverá encaminhar a mensagem recebida. Esta decisão será tomada pela função `demux` com base no cabeçalho do protocolo IP transportado pela mensagem.[1] A mensagem é entregue à função `pop` da sessão. Após lhe ter extraído o seu cabeçalho, a sessão invocará a função `demux` do protocolo TCP, passando como argumento a mensagem recebida. Cada sessão que recebe a mensagem será responsável por actualizar o estado em função dos dados recebidos e por invocar a função `demux` do protocolo acima.

3.2.4 Protocolos virtuais

O x-Kernel define protocolo virtual como um elemento do grafo que se limita a tomar opções sobre o encaminhamento de mensagens (O'Malley & Peterson, 1992). Um protocolo virtual não comunica com os seus pares em grafos remotos, que podem, inclusivamente, não existir. Os protocolos virtuais introduzem um maior dinamismo no comportamento do grafo ao decidirem o encaminhamento individual de cada mensagem a partir dos seus atributos. Torna-se assim possível omitir protocolos redundantes ou que não produzirão efeito em determinadas mensagens, aumentando o desempenho do sistema. Conceptualmente, os protocolos virtuais comportam-se como cláusulas "IF" estrategicamente posicionadas no grafo. As condições concretizadas permitem uma granularidade mais fina dos protocolos que resulta numa diminuição da sua complexidade.

[1]No caso particular do protocolo IP esta informação está contida no campo Protocol (Postel, 1981a).

3.2.5 Delegação

Para além da troca de mensagens, a comunicação entre protocolos pode também ser realizada pela invocação de operações de controlo. Uma operação de controlo não é endereçada a um protocolo em particular. O x-Kernel invoca a operação de controlo no protocolo de nível inferior, que pode opcionalmente delegar noutros a resposta. Uma aplicação utilizando uma pilha TCP/IP determina o endereço IP do sistema onde está a ser executada invocando a operação de controlo adequada. Por o desconhecer, o protocolo TCP delegará na camada inferior (o protocolo IP) a resposta ao pedido.

3.2.6 Gestão de tampões

Para reduzir o número de cópias de memória realizadas nas operações sobre mensagens, o x-Kernel define um tipo abstracto, *tampão*, e um conjunto de funções para as operações elementares (concatenação, corte, divisão, etc.) (Mosberger, 1996). Internamente, o tipo *tampão* foi estruturado sob a forma de uma árvore binária de apontadores para blocos de memória. Esta estrutura reduz as operações comuns a manipulação de apontadores. Em resultado, assiste-se à eliminação de praticamente todas as operações de cópia de memória, que penalizam fortemente o desempenho das plataformas de suporte à distribuição (Druschel *et al.*, 1993).

3.2.7 Comentários

Os resultados de desempenho apresentados pelo x-Kernel, muito competitivos com concretizações monolíticas dos mesmos protocolos, mostram que a modularidade dos protocolos de redes de dados, conceptualmente universalizada com o modelo OSI pode também ser eficiente na concretização. No entanto, estes resultados só são atingíveis quando o ambiente oferece ferramentas devidamente optimizadas para as operações mais comuns e consumidoras de recursos.

O x-Kernel satisfaz estes requisitos pelas seguintes razões:

- oferece bibliotecas optimizadas para as operações mais comuns dos protocolos: manipulação de mensagens e projecções entre identificadores e sessões;
- reduz as mudanças de contexto na troca de mensagens;
- reutiliza objectos dinâmicos, nomeadamente sessões e processos para encaminhamento de mensagens.

A concretização dos protocolos torna-se também mais simples com a sua distinção das instâncias. O conceito de sessão permite uma mais fácil detecção de pontos críticos e de erros, ao localizar e individualizar a informação de estado que os protocolos têm que manter.

O encaminhamento de mensagens é realizado por protocolos e por protocolos virtuais. Este modelo não oferece ainda toda a transparência e modularidade possível. O encaminhamento de mensagens, definido por cada protocolo, é baseado num identificador apresentado pelo protocolo de nível superior. Este modelo pressupõe que os protocolos:

1. conhecem a dimensão do identificador utilizado pelo protocolo de nível superior, pois só desta forma poderão programar convenientemente o seu cabeçalho.[2]
2. têm o seu identificador previamente estabelecido, e que ele é único no conjunto de composições possíveis dos protocolos de nível inferior.

De acordo com a definição apresentada em 2.1, estes requisitos beneficiam a configurabilidade em detrimento da reusabilidade: são suficientes para combinações estáticas, mas pobres para combinações dinâmicas em que as propriedades são adicionadas em função do interesse do programador de aplicações. Nestas, a extrema independência entre protocolos, que leva cada um a ignorar a existência dos restantes, pode gerar situações de conflito não solúveis sem a alteração do código dos protocolos. Os protocolos virtuais têm uma associação forte à composição em que são utilizados e

[2] Em alternativa é possível recorrer a soluções mais pesadas do ponto de vista de desempenho, como a utilização de delimitadores.

são por isso de difícil reutilização. Um protocolo virtual que omita do caminho de uma mensagem um protocolo de fragmentação p tem que saber a dimensão dos fragmentos gerados por p.

Para manter o encapsulamento, os grafos de protocolos têm que respeitar algumas restrições. Em particular, sem que exista um conhecimento da configuração do grafo, não é possível concretizar protocolos que tomem opções de encaminhamento de mensagens no sentido descendente.

O encaminhamento de mensagens baseado em decisões dos protocolos ou dos protocolos virtuais é adequado se as camadas de nível superior, em particular, a aplicação, não forem consultadas. No entanto, quando a aplicação envia dados de diversos tipos, por exemplo, informação multimédia, onde não é possível às camadas fazerem interpretação semântica sobre o que transportam, cabe a esta a responsabilidade da sinalização do caminho a ser percorrido. O x-Kernel não oferece nenhum mecanismo que permita a satisfação deste requisito.

3.3 Coyote

O Coyote (Bhatti *et al.*, 1998) é um sistema de suporte à composição em que, ao contrário das plataformas de composição vertical, as mensagens podem ser processadas em paralelo pelos diversos protocolos. Este modelo incentiva a existência de interacções complexas e a partilha de memória entre protocolos, sacrificando a reusabilidade.

A figura 3.2 apresenta um exemplo de composição no Coyote.

3.3.1 Descrição

A plataforma é orientada aos eventos. Cada protocolo regista no núcleo do sistema o conjunto de eventos que pretende receber e a função que os trata. Por omissão, um evento é entregue concorrentemente a todos os protocolos que o registem. Quando são recebidas na plataforma, as mensagens são colocadas num conjunto não ordenado,

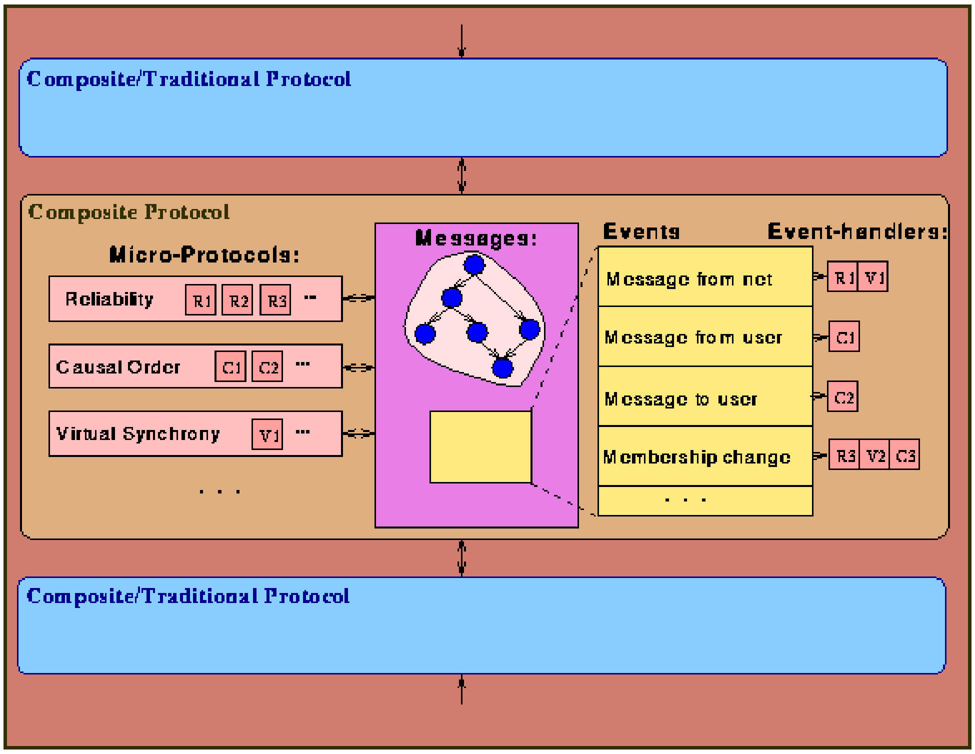

Figura 3.2: Composição de protocolos no Coyote (Bhatti *et al.*, 1998).

acessível a todos os protocolos. Nesse momento, o Coyote gera um evento notificando a chegada de uma nova mensagem, que será entregue a todos os protocolos que o tenham registado. Os protocolos dispõem de duas ferramentas para comunicarem entre si: memória partilhada e eventos. Uma mensagem abandona o Coyote quando todos os protocolos manifestarem a sua concordância, o que é individualmente realizado pela activação de um digito binário associado à mensagem. Cada mensagem tem associada uma zona de memória partilhada contendo os seus atributos. Os atributos são sintetizados (à saída) e extraídos (à chegada) de um cabeçalho único adicionado pelo Coyote no momento em que a mensagem sai ou entra na plataforma.

3.3.2 Comunicação entre protocolos

Para além dos eventos predefinidos pela plataforma, qualquer protocolo pode, em tempo de execução, definir, reconfigurar e gerar eventos. A associação entre os eventos e as funções que os tratam é estabelecida em tempo de execução e pode ser alterada dinamicamente. Na configuração dos eventos são definidos dois parâmetros:

Sincronia O evento é síncrono se a função que o gerou fica suspensa até que as funções gestoras de eventos registadas terminem a sua execução. Pelo contrário, é assíncrono se a função que gerou o evento continuar a sua execução imediatamente.

Concorrência O evento pode ser tratado em paralelo por todas as funções tratadoras de eventos que o registaram, ou sequencialmente por cada uma delas.

A plataforma não oferece qualquer facilidade para o controlo de concorrência, relegando para os programadores, a sua concretização.

3.3.3 Comentários

Quando comparado com outras plataformas (como por exemplo o x-Kernel, apresentado na secção 3.2) constata-se uma diferença profunda na filosofia de tratamento das mensagens. Ao invés de cada mensagem ser tratada sequencialmente por uma camada, todas as camadas tratam, simultaneamente, a mesma mensagem. Deixa por isso de fazer sentido usar as expressões "composição hierárquica" e "composição vertical", associadas que são à existência de uma ordem entre os protocolos. Expressões mais adequadas a este modelo serão "composição paralela" ou "composição horizontal´´, onde fica representado o processamento concorrente da mensagem.

A flexibilidade da comunicação entre protocolos é conseguida pela tolerância à criação de dependências entre eles e que prejudicam a sua reusabilidade. O suporte à utilização de memória partilhada, a possibilidade de cada programador poder, em tempo de execução, definir novos tipos de eventos e a ordenação de gestores no tratamento dos eventos sequenciais, por exemplo, incentivam o estabelecimento de dependências que diminuem a possibilidade de reutilização de um protocolo num outro contexto. No entanto, ao favorecer a modularidade, facilita os processos de reconfiguração, aumentando por isso, a configurabilidade dos protocolos.

A concepção do Coyote apresenta alguns pontos que favorecem o seu desempenho. A facilidade com que pode ser partilhada informação entre protocolos e o conceito de atributos de mensagens permitem eliminar informação redundante adicionada por várias camadas num modelo estritamente hierárquico. Em consequência existe uma redução da quantidade de dados enviados que resulta na aceleração das operações de cópia em memória, na eliminação de operações redundantes e na diminuição do tempo de transporte.

3.4 Horus

O Horus (van Renesse *et al.*, 1996) é uma plataforma de suporte à comunicação em grupo.

A pilha é definida em tempo de execução, adaptando-se assim mais facilmente às condições e à aplicação que a irá utilizar. Uma única instância do Horus pode suportar simultaneamente diversas aplicações, com pilhas diferentes, e onde cada instância de uma camada terá o seu estado próprio (van Renesse *et al.*, 1995a; van Renesse *et al.*, 1994).

O Horus pode ser executado no espaço de endereçamento do utilizador, no núcleo do sistema operativo ou dividido entre ambos. Fica assim ao cuidado do utilizador a ponderação dos factores enunciados na secção 2.3.4. As aplicações podem aceder transparentemente ao Horus através do seu encapsulamento nas primitivas de comunicação UNIX (*sendto*, *recvfrom* e *select*). Foi também desenvolvido uma interface para a linguagem de programação Tcl/Tk (van Renesse *et al.*, 1996; van Renesse *et al.*, 1995a).

3.4.1 Modelo

A plataforma é totalmente orientada aos eventos. Cada camada é identificada por uma cadeia de caracteres ASCII. No momento de inicialização da pilha, as camadas registam as suas funções gestoras de eventos e recebem argumentos de configuração.

A plataforma dispõe de escalonadores dinâmicos de processos ligeiros e de memória, que mantêm o equilíbrio do consumo de recursos entre as diversas pilhas, geridas concorrentemente.

O Horus identifica as seguintes entidades (van Renesse *et al.*, 1996):

Pontos de comunicação, que modelam a entidade comunicante. São eles que enviam e recebem mensagens. A sua projecção num mecanismo concreto (por exemplo, um processo ou um porto) é dependente da aplicação. Os pontos de comunicação são identificados por um endereço, utilizado para a filiação nos grupos. Um processo pode deter vários pontos de comunicação, cada um com a sua pilha de protocolos.

Grupos, utilizados para manter o estado local do protocolo num ponto de comunicação. Cada grupo é identificado por um endereço de grupo, utilizado pelos pontos de comunicação para o endereçamento das mensagens. Cada um destes objectos tem associada uma vista, ou seja, uma lista dinâmica dos pontos de comunicação filiados no grupo. Note-se que os objectos grupo são locais a cada instância da pilha donde, à partida, diferentes ponto de comunicação podem manter diferentes vistas no mesmo instante. A uniformização destas, se pretendida, será conseguida através da utilização de protocolos apropriados. Um ponto de comunicação pode estar filiado, simultaneamente, em diversos grupos. São estes objectos que mantêm o estado das camadas da pilha, assegurando a sua independência de outras instâncias.

Mensagens, estruturas locais cuja interface contém operações de adição e extracção de cabeçalhos. As mensagens são passadas entre camadas por apontadores de memória, eliminando a necessidade de operações de cópia.

3.4.2 Tratamento de eventos

A Horus Common Protocol Interface (HCPI)[3] descreve a assinatura das funções que todos os protocolos têm que concretizar e que estabelecem a ligação entre eles e o núcleo da plataforma. As funções do HCPI podem ser categorizadas em envio, recepção e estabilidade de mensagens e filiação em grupos.

Um dos objectivos do HCPI foi evitar o recurso frequente a operações de controlo, como sucede no x-Kernel, e que aumenta a dependência entre camadas (van Renesse *et al.*, 1996), diminuindo a reusabilidade.

3.4.3 Construção de pilhas irregulares

No Horus, existem dois protocolos que reúnem várias "sub-pilhas", transformando a composição numa árvore. Embora resolvendo problemas específicos da comunicação em grupo, estes protocolos provam a insuficiência das pilhas como modelo único das plataformas de suporte à comunicação.

3.4.3.1 Escalabilidade

Grande parte dos protocolos de comunicação em grupo sofre de problemas de escalabilidade: o desempenho do conjunto deteriora-se com o crescimento do número de membros. Uma solução possível é a hierarquização dos grupos de grande dimensão, criando estruturas de filiação em árvore.

No Horus, as estruturas de filiação hierárquica são geridas por protocolos PARCLD (*client-server membership*). Sob cada uma destas camadas são definidas duas "sub-pilhas", como apresentado na figura 3.3a). O protocolo direcciona as mensagens para cada uma das sub-pilhas, dedicando uma à comunicação do ponto com os seus pares e outra à comunicação com os pontos de nível inferior na árvore (van Renesse *et al.*, 1995b). A utilização deste protocolo é transparente para os restantes, o que favorece

[3] Em (van Renesse *et al.*, 1994) a HCPI é identificada como *Uniform Group Interface* (UGI).

a reusabilidade, mas força o protocolo PARCLD a interpretar todas as mensagens que circulam nas pilhas.

3.4.3.2 Grupos ligeiros

Ao contrário do protocolo PARCLD, o objectivo dos grupos ligeiros (*Light-Weight groups*) é permitir a divisão da parte superior da pilha (figura 3.3b). Esta camada cria conjuntos de grupos que partilham parte das propriedades de um grupo base do Horus mas que, acima da camada LWG suportam individualmente outras propriedades. O objectivo dos grupos ligeiros é aumentar o desempenho do Horus. Grupos com uma filiação semelhante podem partilhar algumas propriedades como a detecção de faltas e a ordenação, reduzindo assim o consumo de recursos (Rodrigues *et al.*, 1996).

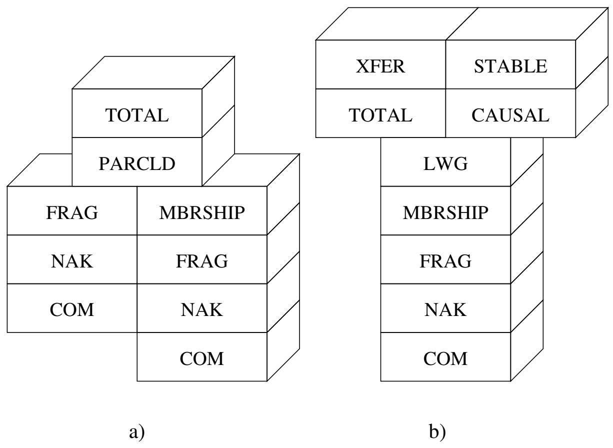

Figura 3.3: Possibilidades de construção de pilhas irregulares no Horus

3.4.3.3 Aceleração de mensagens

Parte das camadas de uma pilha vocacionada para a comunicação em grupo são úteis apenas durante alguns períodos de tempo. A camada que assegura a unicidade das vistas, por exemplo, só se torna necessária quando é suspeitada a falta de um dos membros, ou no momento em que novos membros são adicionados ao grupo. Em condições ideais (por exemplo, quando não ocorrem perdas de mensagens e não é detectada falha em nenhum dos membros do grupo), o cabeçalho e o tempo de processamento dessas camadas consomem desnecessariamente recursos do sistema.

A camada FAST (*Message acceleration*) (van Renesse *et al.*, 1996) estabelece um encaminhamento inteligente, em função das condições do ambiente. Enquanto as condições permanecem ideais, ela assegura a ordenação FIFO das mensagens e evita que a mensagem visite um conjunto de camadas pré-definido e que não lhe acrescentam propriedades úteis. Quando são detectadas anomalias, esta camada restabelece o circuito usual de circulação de mensagens. O protocolo FAST pode ser visto como um caso particular dos protocolos virtuais propostos pelo x-Kernel e descritos na secção 3.2.

3.4.4 Comentários

O Horus é uma plataforma de suporte à distribuição com grande flexibilidade e que relega para o utilizador algumas opções, como o ambiente de execução (no núcleo ou ligado ao programa como uma biblioteca) e a própria configuração da pilha, determinada em tempo de execução.

Ao contrário do x-Kernel, o Horus impede a existência de operações de controlo o que permite aumentar a modularidade. Do ponto de vista sintáctico as camadas podem ser dispostas livremente na pilha, independentemente dos resultados obtidos. No entanto, a opção de catalogar exaustivamente os eventos possíveis limita as possibilidades de extender a plataforma para além do domínio previsto pelos autores. Esta perspectiva confronta-se com a adoptada no Coyote, onde a grande maioria dos eventos é definida pelos protocolos. Surgem assim duas posições antagónicas: o Horus favorece a reusabilidade limitando, pela lista de eventos possíveis, os protocolos que poderão vir a ser desenvolvidos. O Coyote incentiva a concretização de novos protocolos sacrificando a reusabilidade dos existentes.

É evidente a tendência para uma composição estritamente vertical. A necessidade de pilhas não regulares é reconhecida apenas de forma limitada. As camadas PARCLD, FAST e LWG são apresentadas como excepções que contornam os princípios da plataforma, impedindo outros desenvolvimentos nesse sentido.

No entanto, as situações identificadas por estas camadas são apenas algumas das necessidades que podem ser mais facilmente concretizadas em plataformas que suportem também alguma forma de composição horizontal.

3.5 Ensemble

O Ensemble (Hayden, 1998) é a continuação do projecto Horus, desta vez utilizando a linguagem Ocaml, uma extensão ao ML. Ambas são linguagens funcionais, fortemente tipadas, com tratamento de alto nível de estruturas e gestão automática de memória. O Ocaml dispõe de ferramentas de serialização, de um gerador de *bytecode* e de um compilador.

3.5.1 Modelo

O Ensemble não admite alterações ao modelo estritamente vertical da pilha de protocolos. Esta estrutura é encimada por uma camada denominada *Top* e terminada numa camada *Bottom*, que são necessariamente, os únicos pontos de interacção entre as camadas e o núcleo que as suporta. Os eventos são a única forma de comunicação entre camadas.

A composição segue o modelo do Horus: existe uma interface única que terá que ser respeitada por todas as camadas, o que torna qualquer composição de protocolos sintacticamente correcta. Nas relações entre camadas, são asseguradas duas invariantes: os eventos são passados entre camadas respeitando uma ordenação FIFO e nenhuma camada processa dois eventos em simultâneo.

Uma característica particular das pilhas Ensemble é o posicionamento da aplicação. Ao contrário dos modelos clássicos, a aplicação não é posicionada no topo da pilha mas paralelamente a esta (figura 3.4). O envio e recepção de mensagens é preferencialmente realizado pela camada *Appl* que dispõe de uma interface própria para entrega dos eventos mais relevantes à aplicação. No entanto, a aplicação é livre de interagir com as restantes camadas bastando para tal que estas disponibilizem a interface necessária. A camada *Appl* é posicionada o mais abaixo possível na pilha para retirar do caminho das mensagens trocadas pelas aplicações todas as camadas que não lhe são directamente relevantes. Isto diminui substancialmente o tempo de transmissão (van Renesse *et al.*, 1997).

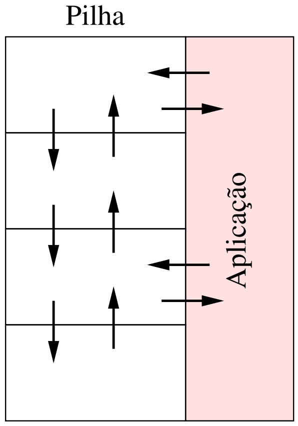

Figura 3.4: Interacção entre a pilha Ensemble e a aplicação (van Renesse *et al.*, 1997)

As instâncias das pilhas estão associadas às vistas: a instalação de uma nova vista faz com que o núcleo da plataforma gere uma nova instância de cada protocolo, criando uma nova pilha. Para além de simplificar a codificação das propriedades é esta característica que permite ao Ensemble a mutação da configuração da pilha durante a sua execução. Para evitar atrasos desnecessários, o processo de terminação de uma pilha pode ser executado simultaneamente com a instalação da nova (Hayden, 1998).

3.5.1.1 Eventos

Os tipos de eventos, exclusivamente definidos pela plataforma, são estruturas de dados. O encapsulamento é mais rígido no Ensemble do que nas restantes plataformas apresentadas: a única forma de comunicação das camadas com o exterior é por eventos. Um exemplo extremo desta situação é o pedido de um temporizador, realizado pelas camadas através do envio para a pilha de um evento *Timer* no sentido descendente e propagado até abandonar a camada *Bottom*, momento em que é entregue ao núcleo. Quando é atingido o tempo para o qual o alarme tinha sido solicitado, a plataforma introduz na pilha um novo evento *Timer* desta vez no sentido ascendente.

As mensagens são a única forma de comunicação entre pilhas. O processo de conversão das estruturas de dados é da responsabilidade do núcleo.

3.5.2 Optimizações

O fundamento teórico das optimizações utilizadas no Ensemble passa pela abstracção dos protocolos ao conceito matemático de função, tal como apresentado na secção 2.1. Torna-se assim possível utilizar a avaliação parcial de funções para, conhecido o evento e parte do estado do protocolo, determinar antecipadamente parte do resultado da sua execução. Desta forma, reduz-se o esforço computacional da execução do protocolo, diminuindo o tempo de processamento do evento e, consequentemente, a latência das mensagens. A avaliação parcial pode ser extendida à operação de composição: conhecida parte do resultado de execução de um protocolo é possível avaliar parcialmente aqueles que com ele estão compostos, resultando numa execução final mais eficiente da função composta (van Renesse *et al.*, 1997).

3.5.2.1 Trilhos de eventos

A unidade básica de optimização utilizada pelo Ensemble é o "trilho de eventos": a sequência de operações executadas pelas diversas camadas em resultado da ocorrência de um evento (Hayden, 1998).

O objectivo das optimizações é a diminuição do tempo de execução dos trilhos de eventos mais comuns, através da definição de um "gestor de evento" que reune, numa única função, o código executado em resposta ao evento por todos os protocolos. O gestor de eventos será executado se o *predicado de caso comum*, que avalia o tipo de evento e o estado das camadas, for verdadeiro. Para a sua construção, o gestor de evento requer que o código fonte das camadas seja anotado pelos programadores de protocolos.

As optimizações aplicadas ao gestor de eventos permitem eliminar código não utilizado, tornado visível pela criação de constantes, como o tipo de evento e remover das sequências de eventos mais frequentes todo o processo de gestão das filas entre camadas.

3.5.3 Maestro

O Maestro (Birman *et al.*, 1997) é um integrador de protocolos, podendo coordenar composições suportadas por diferentes plataformas, como *sockets* UDP (Postel, 1980), Collaborative Computing Transport Layer (CCTL) (Rhee *et al.*, 1997) e pilhas Ensemble. Aplicações com requisitos complexos de comunicação podem utilizar o Maestro como facilitador da integração, assegurando desta forma, um conjunto de propriedades adicionais, como a atribuição de prioridades a mensagens, segurança e informação consistente de filiação nos grupos.

A figura 3.5 apresenta uma possível combinação de protocolos no Maestro. A aplicação utiliza uma pilha Ensemble para disseminar a informação de controlo aos membros.

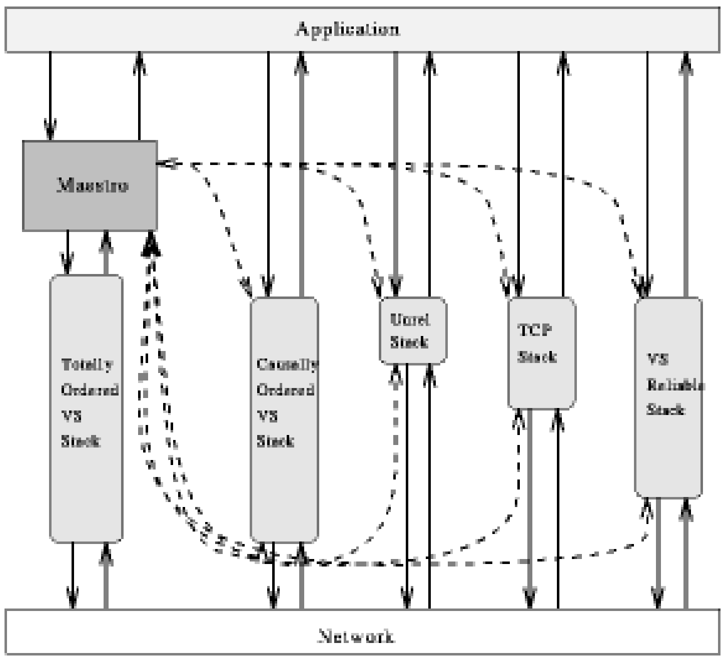

Figura 3.5: Combinação de protocolos no Maestro (Birman *et al.*, 1997).

O aspecto mais relevante do Maestro é proporcionar à aplicação uma visão integrada de um conjunto de protocolos não relacionados: a pilha de coordenação dissemina informação de filiação por todos os membros, garantindo unicidade das vistas.

3.5.4 Comentários

O Ensemble apresenta duas características particulares: composição estritamente vertical e encapsulamento forte dos protocolos. Esta posição é justificada por (Hayden, 1998):

- permitir alguns tipos de operações de optimização;
- facilitar a verificação formal dos protocolos;
- simplificar a depuração da aplicação e dos protocolos;

Por outro lado, a utilização estrita da pilha como forma de composição, coloca em evidência as deficiências do modelo para requisitos que ultrapassem as aplicações convencionais.

Os eventos assumem-se no Ensemble como a única forma de alteração de estado das camadas. Nas plataformas analisadas anteriormente, esta missão era repartida com o núcleo de suporte aos protocolos. O número de situações em que um temporizador poderá ser partilhado por mais de uma camada é muito reduzido e a ocorrência propositada destas situações é um factor que diminui a reusabilidade dos protocolos. Apesar de irrelevantes para as restantes camadas, os dois eventos percorrem todo o caminho entre o fundo da pilha e a camada que solicitou o temporizador, desperdiçando recursos.

No Ensemble, a pilha tem que ter um formato estritamente hierárquico, ao contrário do Horus, onde embora de forma limitada, havia algumas possibilidades de paralelização de camadas. Em consequência, o serviço de grupos ligeiros, por exemplo, foi realizado na zona de interface entre a camada Appl e a aplicação. Esta solução, para além de contra intuitiva, limita as possibilidades de configuração do serviço: todos os grupos ligeiros partilham rigorosamente as mesmas propriedades da pilha (Miranda *et al.*, 1998).

O Maestro apresenta-se como uma forma de colmatar a inflexibilidade da composição. No entanto, a solução é apenas limitada. A informação disseminada é insuficiente quando surgem outros requisitos de integração. Nesse caso, é à aplicação que cabe a coordenação necessária para a obtenção das propriedades desejadas. A sincronização de dados recebidos por diferentes pilhas, por exemplo, terá que ser realizada pela aplicação.

No entanto, as restrições são um factor importante para o desempenho e para a validação formal dos protocolos. As expectativas quanto à possibilidade de criação de um optimizador automático de código, estão fortemente associadas ao encapsulamento total do estado das camadas e aos limitados pontos de interacção com o exterior de que dispõem.

Para contornar a rigidez do modelo, o Ensemble apresenta um núcleo "forte", capaz de interpretar a semântica dos eventos e interveniente em alguns dos algoritmos. Os algoritmos de mudança de vista e de troca de protocolos são exemplos destas características: no primeiro, o núcleo interpreta o evento de mudança de vista, respondendo com a construção de uma nova instância da pilha; no segundo, é o próprio núcleo que, ao longo da execução do Protocol Switch Protocol (PSP) insere alguns eventos na pilha.

Um núcleo forte é uma característica não observável, por exemplo, no x-Kernel e no Coyote. A impossibilidade de o utilizador definir eventos é crucial. Se, à semelhança do Coyote, o Ensemble oferecesse esta facilidade, seria bastante mais difícil o núcleo manter a capacidade de intervenção de que dispõe. Constata-se assim que, no Ensemble, à utilização de uma modularidade mais estrita e que favorece a reusabilidade, se contrapõe um núcleo limitador do conjunto de protocolos concretizáveis.

Utilização do ML A utilização da linguagem ML é justificada pelas facilidades de alto nível que apresenta e pela fácil adaptação aos modelos formais de verificação de protocolos. O ML tinha já sido experimentada no Horus (van Renesse *et al.*, 1995a), onde era utilizado como linguagem de referência e especificação. O desempenho da linguagem levava contudo a que as versões altamente optimizadas das camadas fossem concretizadas nas linguagens C ou C++. A prática reforça a perspectiva anterior: a latência do código optimizado é aproximadamente 40% inferior se escrito na linguagem C. Do ponto de vista de concretização, os resultados obtidos são opostos: funcionalidades idênticas, codificadas nos dois sistemas, usam sete vezes menos linhas de código no Ensemble (Hayden, 1998).

3.6 Bast

O Bast (Garbinato *et al.*, 1996) utiliza a herança como ferramenta principal da composição de protocolos. A plataforma foi codificada numa linguagem orientada aos objectos: o Smalltalk.

3.6.1 Modelo

O Bast divide-se em duas componentes: a biblioteca de protocolos (o Bast) e a plataforma de suporte à execução dos protocolos (Bastet).

O Bast utiliza a herança simples como ferramenta de composição dos protocolos: um protocolo que dependa de outro extende a classe que o concretiza. Paralelamente, o Bastet oferece uma interface de alto nível para os programadores de aplicações, escondendo a complexidade de lidar directamente com os protocolos.

A plataforma utiliza o padrão de desenho *Strategy* (Gamma *et al.*, 1995) para distinguir os protocolos dos algoritmos que o concretizam. São por isso definidas duas classes base (Garbinato & Guerraoui, 1997).

A classe `ProtoObject` é a raiz da hierarquia de protocolos. Classes descendentes de `ProtoObject` modelam a interface de um protocolo e realizam, em tempo de compilação, a ligação aos protocolos de que dependem. A ligação pode ser feita através de herança e/ou por instanciação de outras sub-classes de `ProtoObject` que disponibilizem as propriedades necessárias.

A classe `ProtoAlgo` modela os algoritmos. As sub-classes de `ProtoAlgo` concretizam algoritmos de um protocolo. Em tempo de execução, a sub-classe de `ProtoObject` pode seleccionar e alternar entre um dos algoritmos possíveis. A utilização de algoritmos está encapsulada nos protocolos: nem o programador de aplicações nem os restantes protocolos dependem do algoritmo em utilização.

Os serviços solicitados ao protocolo são encaminhados para o algoritmo em execução. Este, por sua vez, utiliza o protocolo para encaminhar os pedidos às camadas inferiores. Uma instância de um protocolo pode ter, simultaneamente, diversos algoritmos (iguais ou não) em execução. Cabe ao protocolo identificar univocamente o algoritmo utilizado em cada mensagem, para que o receptor a encaminhe para a instância do algoritmo adequada.

3.6.2 Comentários

O Bast apresenta duas características que o distinguem, em modelo e funcionalidade, dos restantes. A primeira é a utilização da herança como ferramenta de composição, a segunda a utilização do padrão de desenho *Strategy*.

A herança antecipa a ligação dos protocolos para o momento de codificação. Esta solução torna-se vantajosa por dispensar, mais tarde, a validação da composição. No entanto, a utilização de propriedades adicionais só pode ser conseguida através da definição, caso a caso, de novos protocolos, contendo todas as propriedades desejadas. Uma outra característica possibilitada pela herança é o posicionamento da aplicação paralelamente às camadas: uma vez que cada protocolo extende aquele que satisfaz o conjunto de propriedades de que necessita, os métodos disponibilizados pelos protocolos herdados tornam-se também disponíveis para a aplicação. Um protocolo que para ordenar totalmente as mensagens extenda um outro de ordenação FIFO, oferece também à aplicação a interface de ordenação ponto-a-ponto de mensagens. À semelhança do que acontece no Ensemble, à aplicação é possível aceder à interface de todos os protocolos.

A utilização do padrão de desenho *Strategy* torna a plataforma configurável dinamicamente. Um protocolo pode, por exemplo, comutar entre os diversos algoritmos de ordenação total, de acordo com a localização de cada um dos participantes no grupo.

A utilização concorrente de diversas instâncias de protocolos é semelhante ao modelo utilizado no x-Kernel, onde o conceito de sessão corresponde aqui ao de algoritmo. O encaminhamento é também decidido por camada, o que eventualmente, pode ser considerado um consumo desnecessário de recursos por repetir, uma vez para cada protocolo envolvido, a operação de projecção de identificadores em algoritmos.

3.7 Conduit+

O Conduit+ (Hüni *et al.*, 1995) é uma plataforma de suporte à composição que adopta um modelo onde o conhecimento dos programadores de protocolos sobre a plataforma em que realizam o desenvolvimento é muito limitado. O Conduit+ foi desenvolvido na linguagem de programação C++ e utiliza intensivamente os padrões de desenho.

3.7.1 Modelo

A plataforma é composta por dois tipos de objectos: *conduits* e *information chunks* (pedaços de informação).

Um conduit é um componente com dois lados, A e B, aos quais são ligados outros conduits e por onde são trocados os *information chunks*. Não existe relação entre os lados de um conduit e a orientação que apresentam na pilha. Existem quatro classes de *conduits*, com funções distintas (figura 3.6):

Mux, tem um um número variável de lados B. A função de um Mux é reunir zero ou mais conduits do seu lado B e determinar para qual deles encaminha as mensagens recebidas do lado A.

Adapter, não tem lado B. Um Adapter é o ponto de contacto entre uma composição e o exterior. Pode ser utilizado, por exemplo, para receber dados de uma placa de

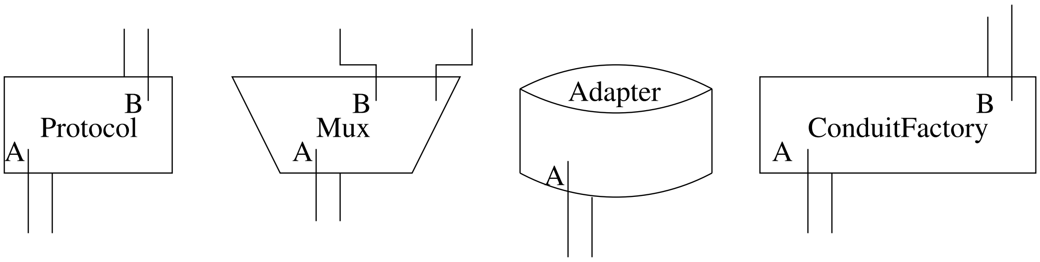

Figura 3.6: Os quatro tipos de conduits

rede, transformando-os em *information chunks* que introduz na composição, ou para fazer a ligação com uma aplicação.

Protocol, tem um lado B. A sua única função é aplicar propriedades às mensagens.

ConduitFactory, tem um lado B. É utilizado conjuntamente com os *Mux*. A sua função é criar novos conduits sempre que o *Mux* a que está associado necessita de encaminhar uma mensagem para um novo lado B.

Nesta plataforma, um protocolo pode ser concretizado por mais que uma unidade de composição. O protocolo UDP por exemplo, seria composto por um conduit do tipo *Protocol*, onde seria calculada a soma de controlo e formatado o cabeçalho, um *Mux* para proceder ao encaminhamento das mensagens para os diversos portos e um *ConduitFactory* para a activação de novos portos.

3.7.2 Padrões de desenho

O Conduit+ utiliza intensivamente padrões de desenho.

O padrão de desenho *Strategy* é utilizado para separar o conceito de conduit do código que o concretiza em cada ponto da composição, dispensando assim a utilização da herança como forma de especialização: o construtor de cada conduit recebe como argumento uma referência para um objecto contendo o algoritmo a executar.

Simultaneamente, o Conduit+ utiliza ainda os padrões de desenho *State*, *Singleton*, *Command*, *Visitor* e *Prototype* (Gamma *et al.*, 1995). Os três primeiros são utilizados em conjunto e fazem com que o estado de cada conduit seja representado por um objecto em separado. A aplicação de um objecto contendo uma mensagem a um objecto que mantém o estado de um conduit, resulta numa nova instância deste último. O padrão de desenho *Visitor* torna as mensagens objectos activos. Os dados são associados a um objecto, o qual tem a responsabilidade de invocar as operações nos conduits que atravessa. O padrão de desenho *Prototype* é utilizado nos ConduitFactories para modelar o tipo de objectos criados para cada novo lado B de um Mux.

3.7.3 Comentários

A necessidade de separar o encaminhamento das mensagens da concretização dos protocolos tinha já sido percebida no x-Kernel , onde os protocolos desempenham um papel semelhante aos conduits Mux e as sessões mantêm o estado de cada instância do protocolo tal como os conduits Protocol. Os ConduitFactory extendem a decomposição ao isolarem também o código de criação de novas sessões de um protocolo, tarefa que o x-Kernel incluía nas funções dos protocolos.

A não existência de correspondência entre os lados de um conduit e a posição em que ele é colocado na composição, torna o Conduit+ mais flexível, permitindo a representação de grafos (e não árvores) de protocolos. Este problema havia sido parcialmente resolvido no x-Kernel pelos protocolos virtuais.

A utilização dos padrões de desenho foi motivada pelo interesse em isolar os protocolos da plataforma. No entanto, não é evidente que esta aproximação traga benefícios. O desempenho fica prejudicado pela criação frequente de objectos, nomeadamente para o estado dos conduits que implica reservas e libertações de memória e operações de inicialização dos objectos.

3.8 GroupZ

À semelhança do Conduit+, o Groupz (Pereira & Oliveira, 1997) também torna as mensagens objectos activos. No entanto, o sistema utiliza as potencialidades da linguagem em que foi codificado (o Java) para concretizar o conceito de "rede activa". Isto é, uma rede em que as entidades transportadas (as mensagens) actuam, na origem e no destino, sobre os protocolos.

A concretização dos protocolos é aberta às aplicações, permitindo que estas os configurem a cada mensagem transmitida. A vantagem deste sistema é permitir que mensagens com requisitos de entrega distintos possam ser transmitidas sobre a mesma pilha. As aplicações podem por isso qualificar as mensagens com critérios semânticos, não conhecidos pelos protocolos.

3.8.1 Modelo

A pilha do Groupz é composta por um conjunto de camadas que invocam métodos do objecto onde a mensagem é transportada. Estes, por sua vez, realizam as alterações ao estado das camadas que reflectem as propriedades que se pretendem aplicadas.

Uma mensagem que não necessite de fiabilidade de entrega, por exemplo, será encapsulada num objecto que não altere o número de sequência do estado da camada responsável por esta propriedade.

3.8.2 Comentários

O Groupz tem uma aproximação oposta à utilizada no conjunto de sistemas analisados anteriormente. As propriedades são seleccionadas individualmente por mensagem e estão codificadas na mensagem em si e não nas camadas. O programador da aplicação tem por isso que ter um conhecimento mais profundo sobre a forma como cada camada oferece as propriedades que concretiza.

Por outro lado, a obtenção de novos conjuntos de propriedades pode passar pela definição de novos "invólucros" de mensagens (da responsabilidade do programador) e não pela adição de novas camadas à pilha (cujo desenvolvimento é tipicamente da responsabilidade de outras entidades). Esta característica diminui as possibilidades de utilização transparente do sistema por programadores não especializados mas expande a flexibilidade do sistema, capaz de se adaptar rapidamente a novos requisitos.

Uma característica do Groupz, não contemplada nas restantes plataformas analisadas, é a possibilidade de as aplicações expressarem para o sistema as características semânticas das mensagens que transmitem.

3.9 Análise comparativa

Os próximos parágrafos sintetizam os pontos mais importantes de cada uma das plataformas analisadas anteriormente. São comparadas três vertentes: modelo, características dos protocolos e comunicação interna ao canal.

3.9.1 Modelo

A tabela 3.1 compara os sistemas analisados no que toca à política de controlo de concorrência utilizada e ao modelo de composição.

Plataforma	Composição	Encaminhamento das mensagens	Processos
x-Kernel	Mista	Camada	Evento
Coyote	Paralela	–	Evento
Horus	Vertical	Núcleo	Misto
Ensemble	Vertical	Núcleo	Opcional
Bast	Mista	Camada	–
Conduit+	Mista	Camada	Evento
Groupz	Vertical	Núcleo	Evento

Tabela 3.1: Modelo das diferentes plataformas analisadas

A coluna *composição* descreve os modelos de grafo de protocolos suportados pelos sistemas. O termo **composição vertical** é utilizado para assinalar as plataformas que suportam pilhas de protocolos enquanto a **composição mista** representa o suporte a árvores de protocolos. O Coyote é o único sistema que pode não estabelecer relações entre os protocolos, processando os eventos em paralelo. Pode-se por isso considerar que suporta **composição paralela** ou **composição horizontal**. A tabela mostra também a forte relação entre o modelo de composição e a entidade que realiza o encaminhamento das mensagens: os sistemas que concretizam a composição estritamente vertical são responsáveis pelo encaminhamento das mensagens enquanto que, nos restantes, o encaminhamento é realizado por cada uma das camadas.

A coluna *processos* compara as políticas de atribuição de fluxos de controlo utilizadas pelos diferentes sistemas. O Ensemble apresenta a particularidade de poder ser executado em diversos modos. Por omissão, utiliza um único processo que trata sequencialmente cada um dos eventos, eliminando assim a necessidade de impor restrições ao controlo de fluxo.

3.9.2 Propriedades dos protocolos

A tabela 3.2 compara os sistemas analisados quanto à forma como permitem a composição dos protocolos.

Plataforma	Momento de composição da pilha	Alteráveis em tempo de execução	Partilha de dados
x-Kernel	Compilação	Não	Não
Coyote	Compilação	Não	Sim
Horus	Execução	Não	Não
Ensemble	Execução	Sim	Não
Bast	Compilação	Não	Não
Conduit+	–	–	Não
Groupz	Compilação	Sim	–

Tabela 3.2: Propriedades dos protocolos nas diferentes plataformas analisadas

No que toca ao momento de composição da pilha, constatam-se duas opções distintas: em tempo de compilação, adoptando uma visão mais estática dos serviços e em tempo de execução, permitindo que a aplicação se adapte, inicialmente, ao ambiente em que será executada. O Ensemble e o Groupz levam este esforço mais longe, oferecendo suporte para a alteração da pilha durante a sua execução. Enquanto o primeiro, orientado à comunicação em grupo, disponibiliza ferramentas para sincronizar a mudança de pilha com uma mudança de vista, no Groupz cabe à aplicação seleccionar, para cada mensagem, a pilha a utilizar. O Coyote é a única plataforma a disponibilizar sem restrições zonas de memória partilhada para a comunicação entre protocolos, diminuindo por isso o encapsulamento dos protocolos.

3.9.3 Propriedades dos eventos

A tabela 3.3 compara a utilização de eventos pelos diferentes sistemas. O termo Evento é aqui utilizado num sentido mais restrito, para representar a comunicação entre camadas de uma composição através de estruturas de dados encaminhadas pelo sistema.

Plataforma	Orientado a eventos	Configuráveis em tempo de execução	Definidos pelo utilizador
x-Kernel	Não	–	–
Coyote	Sim	Sim	Sim
Horus	Sim	Não	Não
Ensemble	Sim	Não	Não
Bast	Não	–	–
Conduit+	Não	–	–
Groupz	Não	–	–

Tabela 3.3: Propriedades dos eventos nas diferentes plataformas analisadas

Perante a definição acima, constata-se que apenas três dos sistemas analisados apresentam suporte a eventos. Os restantes não definem qualquer modelo de comunicação entre protocolos (como é o caso do Bast, Conduit+ e Groupz) ou utilizam invocações bloqueantes de procedimentos, o que acontece no x-Kernel . O mecanismo de herança do Bast dispensa a utilização de eventos: os protocolos são ligados aos de nível inferior em tempo de compilação pelo que o programador está ciente das facilidades oferecidas por cada um deles.

Das plataformas restantes, o Horus e o Ensemble suportam um **modelo de eventos fechado**, onde existe apenas um conjunto limitado e pré-definido de eventos. O conhecimento da semântica de cada evento é depois utilizado para a concretização de optimizações. A utilização do modelo fechado limita a utilização da plataforma em contextos diferentes: o conjunto de eventos definido pode não ser suficiente para expressar as interacções de outros protocolos. Embora mais flexível, uma plataforma que utilize o **modelo de eventos aberto**, como o Coyote, não pode concretizar o mesmo tipo de optimizações de desempenho.

3.10 Sumário

Este capítulo apresenta um conjunto de plataformas modulares de suporte à comunicação. Para cada uma são resumidas as suas características mais relevantes e analisados os aspectos em que apresentam contribuições inovadoras.

4

O Appia

O capítulo é iniciado com a apresentação de dois casos onde a utilização de sistemas de suporte à composição é vantajosa. A particularidade de ambos é a impossibilidade de representar a composição desejada de protocolos através dos modelos de composição em árvore e em pilha. Ficam assim expostas as limitações destes modelos de composição e, consequentemente, dos sistemas que impõem a sua aplicação. Estas limitações são analisadas e utilizadas como motivação para uma proposta alternativa.

4.1 Motivação

Do conjunto das plataformas analisadas, observa-se que os modelos mais comuns de composição são aqueles que agrupam os protocolos em árvore ou em pilha. Esta tendência surge naturalmente dos modelos de referência, nomeadamente os já citados modelo OSI e TCP/IP.

É possível representar por pilhas de protocolos o conjunto de caminhos obtidos entre a raiz de uma árvore e todas as suas folhas. A figura 4.1a) apresenta um composição em árvore para seis instâncias genéricas de protocolos P_A, P_B, P_C, P_D, P_E e P_F. A árvore é composta por três caminhos entre a raiz e as folhas, representados por pilhas na figura 4.1b). No entanto, esta figura omite a partilha de estado que ocorre nas instâncias dos protocolos P_D e P_F. Uma representação mais adequada seria a da figura 4.1c), que evidencía a utilização da mesma instância dos protocolos em diversas pilhas.

A partilha de estado, característica do modelo em árvore, permite a aplicação da

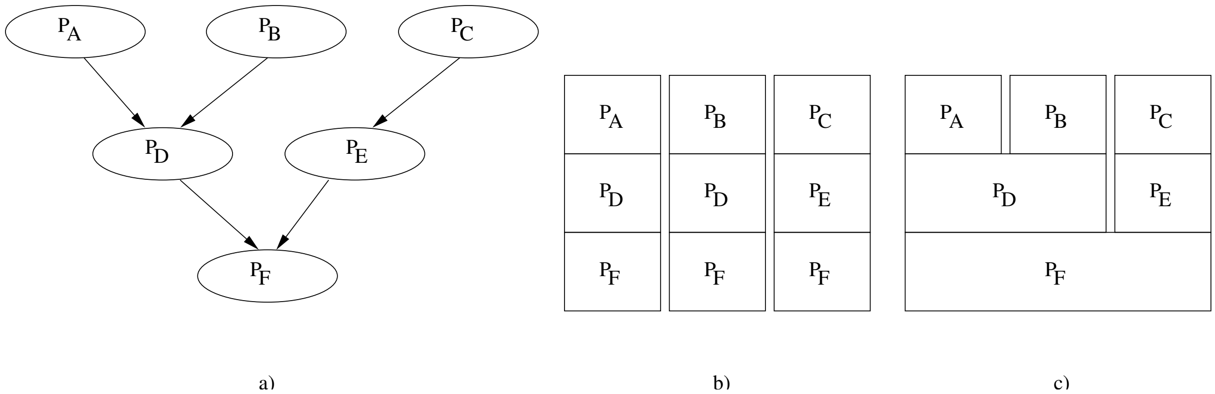

Figura 4.1: Diferentes representações para uma composição de protocolos

mesma instância de uma propriedade a várias pilhas e consequentemente, às mensagens que circulam por elas. No entanto, os sistemas que suportam o modelo de composição em árvore mostram-se insuficientes para a concretização de algumas aplicações, como por exemplo, as que apresentam os dois tipos de requisitos seguintes.

1. A necessidade de partilhar propriedades que não se encontram na zona comum (i.e. abaixo do ponto de separação) da árvore.
2. A necessidade de, em tempo de execução, alterar as propriedades partilhadas.

Esta secção apresenta duas situações em que estes requisitos podem ser encontrados e esboça um modelo de composição adequado. Nas páginas seguintes é apresentado um sistema que o concretiza.

4.1.1 Sincronização de dados com qualidades de serviço independentes

Os diferentes tipos de dados utilizados por uma aplicação multimédia apresentam requisitos de transporte e sincronização diferentes. Assuma-se o exemplo de uma aplicação de vídeo-conferência em que diversos participantes utilizam som, vídeo e mensagens de texto:

garantia de entrega, alguns protocolos de comunicação de vídeo toleram a perda de imagens, no entanto, espera-se que o texto trocado pelos participantes seja entregue a todos os membros;

sincronização, Alguns tipos de dados deverão ser entregues sincronizados entre si à aplicação, como é o caso dos dados de audio e vídeo, enquanto outros (por exemplo, ficheiros transferidos no contexto da mesma aplicação), não devem ser sincronizados com os restantes;

coerência, a falha de um dos participantes na aplicação deve ser informada consistentemente a todos os canais: não faz sentido para os participantes que o canal de transmissão de vídeo assuma a impossibilidade de o contactar enquanto o canal de som continua a receber informação.

Uma aproximação que satisfaz os requisitos de garantia de entrega consiste na utilização de pilhas independentes, uma por cada tipo de dados trocados. A integração dos diferentes tipos de dados e os requisitos de sincronização e coerência teriam que ser concretizados no ponto em que os diferentes canais de comunicação se encontram, ou seja, na aplicação. Esta não é uma solução desejável: a aplicação teria requisitos complexos de comunicação, que desviariam a atenção do programador para problemas de baixo nível.

A sincronização de diferentes tipos de dados numa aplicação multimédia foi já alvo de estudo no âmbito de soluções modulares. Para resolver este problema poderia ser utilizada uma camada de sincronização como a proposta em (Correia & Pinto, 1995). No entanto, esta solução requer a utilização de uma **árvore de protocolos invertida**, ou seja, uma árvore em que diversas pilhas, separadas inferiormente, se reúnem nas camadas mais acima.

O problema da coerência, por sua vez, é solúvel por uma composição em árvore. A informação de detecção de faltas seria gerada por uma camada, comum a todas as pilhas que a partilham, impedindo que a não fiabilidade dos detectores de faltas gerasse situações de informação inconsistente entre os diversos canais de dados.

A união dos requisitos acima citados só poderia ser resolvida adoptando uma composição como a apresentada na figura 4.2. Cada pilha, representada por uma linha vertical que intersecta as camadas que utiliza, partilha algumas propriedades com outras pilhas. As camadas Device, Failure Detector, Intermedia Sync e Application seriam partilhadas por todas as pilhas enquanto as camadas de ordenação (FIFO) e os protocolos específicos dos diferentes tipos de dados, seriam independentes para cada uma. A representação das diferentes pilhas utilizadas produz um efeito semelhante a um diamante. Este tipo de representações é por isso designado por composição em diamante.

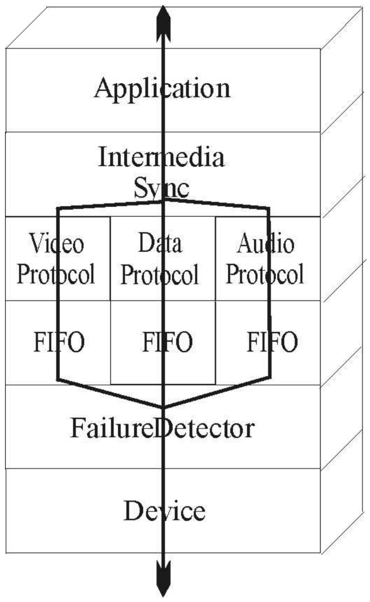

Figura 4.2: Composição em diamante para uma aplicação multimédia

Do conjunto de sistemas analisados, nenhum se adapta facilmente a este modelo, embora em alguns ela seja possível através da criação de camadas não regulares, que sacrificam o modelo inicialmente concebido pelos autores.

No x-Kernel, uma possível solução passaria pela definição de protocolos onde também as sessões tomassem decisões de encaminhamento em função do tipo de dados. Para tal dois requisitos, não convencionais no sistema, teriam que ser satisfeitos: as sessões teriam que ter a capacidade de analisar os cabeçalhos adicionados pelas camadas de nível superior, e cada sessão teria que ser associada a mais de uma camada de nível inferior. A concretização do primeiro sacrificaria totalmente a reusabilidade dos protocolos ao impôr configurações onde os protocolos de nível superior eram conhecidos à partida.

A flexibilidade de composição do Coyote permite suportar uma composição como a apresentada. No entanto, este sistema não dispõe de ferramentas que permitam separar diversas instâncias de execução de um mesmo protocolo. A concretização do exemplo obrigaria à definição de protocolos específicos que separassem os estados das três instâncias do protocolo FIFO, e que fossem capazes de interpretar eventos representativos de cada uma das categorias de dados utilizadas na composição.

O Horus e o Ensemble adoptam um modelo estritamente vertical de composição e inibem por isso composições como a apresentada acima. As excepções previstas no Horus não são suficientemente genéricas para permitir a composição num modelo não concebido pelos autores.

A concretização da configuração da figura no Bast também só seria conseguida contornando os princípios definidos para o sistema. Seria necessário definir protocolos específicos que marcassem as mensagens com o tipo de dados que transportam. À semelhança do que acontece com o x-Kernel , a concretização no Bast confrontar-se-ia com a necessidade de interpretação de cabeçalhos adicionados por outras camadas ou com a redefinição de alguns protocolos que fornecem propriedades comuns, como é o caso do FIFO.

O Conduit+ é o único dos sistemas analisados em que a composição apresentada pode ser realizada sem restrições utilizando as ferramentas disponibilizadas pelo sistema. A separação e reunião das três pilhas do exemplo seria realizada por conduits Mux concebidos para o efeito tal como representado na figura 4.3. A única limitação apresentada pelo sistema passa pela impossibilidade de a aplicação transmitir informação semântica, neste caso o tipo de dados contido nas mensagens, ao Mux de topo, representado na figura por Mux(1). Este conduit seria responsável por garantir o encaminhamento correcto das mensagens que viajam no sentido descendente. À semelhança de algumas das plataformas anteriores, a informação só poderia ser obtida pela análise de cabeçalhos adicionados pelos níveis superiores, nomeadamente pela aplicação.

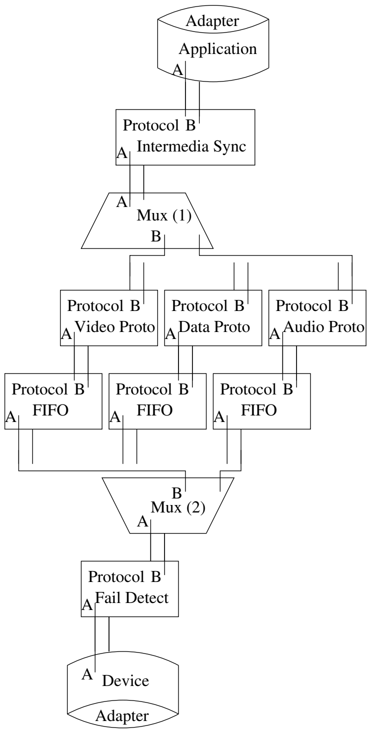

Figura 4.3: Exemplo de composição de protocolos no Conduit+

O suporte às diversas pilhas seria de fácil concretização no Groupz, uma vez que a aplicação pode condicionar, por mensagem, as alterações ao estado dos protocolos. No entanto, o Groupz apresenta capacidades limitadas de expansão no que toca à definição de protocolos. A concretização do protocolo Intermedia Sync teria por isso que ser embebida na aplicação. Do ponto de vista do programador a utilização da plataforma seria também mais complexa uma vez que cabe a este a definição do código a executar pelos protocolos.

4.1.1.1 Comentários

Os sistemas que suportam a composição de protocolos em pilha são claramente insuficientes para resolver um problema com a formulação apresentada anteriormente. Os sistemas onde a composição pode ser realizada em árvore mostram-se mais eficazes, sendo o Conduit+ aquele cuja flexibilidade mais o aproxima dos requisitos. No entanto, mesmo este mostra-se inadequado ao não permitir que a aplicação transmita, para a composição, informação semântica que lhe permita condicionar o encaminhamento. A coluna *Encaminhamento de mensagens* da tabela 3.1 (na página 48) reflecte esta característica: em todos os sistemas que suportam composição em árvore o encaminhamento das mensagens é determinado à medida que elas atravessam a composição. Este modelo é adequado quando as mensagens circulam na sua fase ascendente, isto é, antes da entrega ao receptor: as mensagens foram marcadas pela entidade correspondente com identificadores que permitem, no destino, a determinação do canal adequado.

Em composições complexas como a do exemplo, torna-se necessário que não sejam os protocolos a determinar o encaminhamento das mensagens ou que, pelo menos, seja possível às aplicações condicionar esse mesmo encaminhamento. Nestes casos, o encaminhamento deve satisfazer as propriedades seguintes.

Delegação Um protocolo deve poder delegar em alguma entidade externa o encaminhamento de cada uma das mensagens que trata.

Para fomentar a reusabilidade, os protocolos não deverão realizar interpretação semântica sobre as mensagens. Assim sendo, e apesar de possivelmente participarem em diversas pilhas, não lhes será possível proceder ao seu encaminhamento. Por omissão esta actividade deverá ser delegada numa entidade externa, possivelmente o núcleo de suporte à composição.

Transparência O caminho de uma mensagem pode ser ignorado pelos protocolos.

Para alguns protocolos poderá ser irrelevante o caminho percorrido pela mensagem. Nesse caso, a propriedade por ele concretizada será partilhada por todas as mensagens que circulem por qualquer um dos canais a ele associados.

Opacidade Um protocolo deverá ter acesso aos identificadores de caminho, tratando-os como valores opacos.

Estes identificadores deverão ser únicos e reutilizáveis por forma a permitir o envio, por qualquer camada, de novas mensagens. Assegura-se assim que qualquer camada poderá, de forma transparente, enviar mensagens causalmente relacionadas com as recebidas sem ter necessariamente conhecimento do número ou das características dos canais a que está associado.

Nenhum dos requisitos anteriores deverá ser obrigatório. O sistema deverá providenciar as ferramentas que permitam a qualquer camada, e também à aplicação, ter o conhecimento dos canais a que estão associados, o que compromete a reusabilidade. No entanto, esta nem sempre é desejável. No exemplo, não fará sentido que a camada de sincronização ignore os canais a que está associada. É esta facilidade que permite também à aplicação qualificar semanticamente os dados que envia, associando-os a canais distintos, com características por ela bem conhecidas.

4.1.2 Grupos ligeiros

Os algoritmos de filiação e detecção de faltas utilizados nos sistemas de comunicação em grupo consomem recursos não desprezávis. Quando um processo participa simultaneamente em vários grupos e a filiação é semelhante, o consumo de recursos pode ser reduzido se forem definidos dois níveis de grupos: grupos de nível do utilizador (grupos ligeiros) e grupos de baixo nível que correspondem às entidades definidas pela plataforma de suporte à comunicação. Os grupos ligeiros são entidades transparentes para o sistema. Uma camada da composição projecta os grupos ligeiros utilizados pelo processo em grupos de baixo nível com filiação semelhante. As propriedades de um grupo de baixo nível definidas abaixo da camada de suporte aos grupos ligeiros são partilhadas, promovendo a sua reutilização e um menor consumo de recursos. Idealmente, a projecção deve ser dinâmica: um grupo ligeiro pode mudar de grupo de baixo nível em tempo de execução se tal se mostrar vantajoso.

A figura 4.4 apresenta uma configuração de uma pilha onde a camada LWG define dois grupos ligeiros utilizando propriedades de ordenação distintas. Todas as propriedades definidas abaixo desta camada são partilhadas por ambos os grupos.

Os grupos ligeiros são facilmente modelados por composições em árvore. A sua primeira concretização surgiu no contexto do Horus (Rodrigues *et al.*, 1996). A definição da composição em tempo de compilação apresentada pelo x-Kernel , pelo Conduit+ e pelo Bast limita as possibilidades de configuração dinâmica do protocolo.

O Horus concretiza o protocolo LWG como uma das excepções à composição em

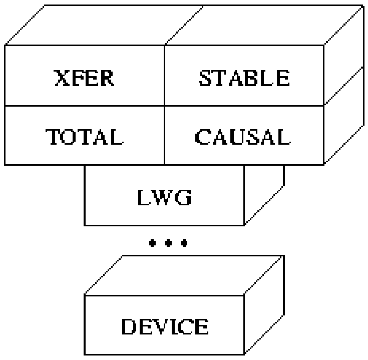

Figura 4.4: Composição utilizando grupos ligeiros

pilha. Não é possível generalizar o modelo a protocolos com requisitos semelhantes. O Ensemble, seu sucessor, abandonou por completo estas excepções, limitando fortemente as possibilidades de concretização do protocolo (Miranda *et al.*, 1998).

4.1.3 Comentários

As limitações apresentadas pelas plataformas em estudo na resolução destes dois exemplos vêm demonstrar a insuficiência dos modelos clássicos de composição, nomeadamente em árvore e em pilha.

Identificam-se quatro requisitos, que não foram até agora reunidos para permitir a solução de composições complexas: ligação tardia, facilidade de a aplicação condicionar o encaminhamento das mensagens, criação dinâmica de instâncias de protocolos e composições irregulares.

Os três primeiros são proporcionados individualmente por alguns dos sistemas estudados. O Coyote oferece uma versão extrema do último ao não impôr a hierarquização dos protocolos.

Neste capítulo será apresentada uma plataforma que suporta composições hierárquicas irregulares de protocolos, definidas em tempo de execução. Este sistema adapta-se facilmente para resolver os problemas acima mencionados, mantendo as características mais importantes dos sistemas analisados.

4.2 Modelo

Nas plataformas discutidas anteriormente, verifica-se que aquelas que suportam a ligação tardia das camadas, assumem que a composição será realizada pelo modelo mais restrito de pilha de protocolos. A necessidade de suportar a ligação tardia em grafos de protocolos foi justificada pelos exemplos apresentados em 4.1.

O *Appia* [1] é um sistema de suporte à comunicação desenvolvido em Java por uma equipa do Departamento de Informática da Faculdade de Ciências. O seu objectivo é preencher a lacuna existente no conjunto das plataformas de suporte à comunicação analisadas, oferecendo a ligação tardia entre protocolos para composições complexas. Paralelamente, foi também tida em conta a necessidade de assegurar que a flexibilidade oferecida não prejudica substancialmente o desempenho do sistema.

No *Appia*, a composição de protocolos tem duas vertentes distintas. A vertente estática modela a Qualidade de Serviço (QdS), enquanto a vertente dinâmica a concretiza. O objectivo desta secção é descrever a forma como ambas se complementam para prestar o serviço.

Cada protocolo é decomposto numa camada, representante estático, que descreve as suas interacções com o exterior e numa sessão, dinâmica, que o concretiza. A plataforma é totalmente orientada aos eventos, pelo que as camadas enumeram os eventos relevantes para os protocolos e as sessões contêm o código que gera e trata esses mesmos eventos.

Uma pilha de camadas, denominada QoS, descreve uma composição de protocolos: enumera e estabelece a ordem pela qual os eventos visitarão os protocolos. Um canal é a instanciação de uma QoS: reúne, pela ordem estabelecida, sessões dos protocolos declarados na QoS. O canal assegura a ordenação FIFO dos eventos entre sessões.

Estes conceitos estão relacionados na tabela 4.1.

[1] A rede viária ligou entre si os mais longíquos cantos do poderio romano, mantendo a unidade do Império e da civilização. A construção dessa imensa rede, que percorreu o mundo como se fosse uma única cidade, consagrou aos imperadores o poderio político e económico. Facilitou a administração das províncias e a deslocação das tropas tendo um papel estratégico tão importante como o económico. A *Via Appia* foi mandada construir pelo censor Appius Claudius Caecus em 312 A.C. e, quando concluída, ligou Roma ao porto de Brundisium, no Mar Adriático.

Conceito	Tipo	Função
Camada	Estático	Descreve as interacções do protocolo com o exterior
Sessão	Dinâmico	Concretiza as interacções do protocolo com o exterior
QoS	Estático	Pilha de camadas
Canal	Dinâmico	Pilha de sessões

Tabela 4.1: Relações entre QoS, Canal, Camada e Sessão no *Appia*

4.2.1 Composição

O *Appia* suporta ligação tardia: as QoSs e os canais são definidos em tempo de execução. Uma QoS é definida por um vector das camadas que a irão compor. Um canal é definido através de um pedido realizado à QoS respectiva. A figura 4.5 representa este processo. Após a criação, um canal não é mais que um vector de sessões vazio. Cada uma das posições deste vector deverá ser preenchida com uma sessão da camada declarada na QoS. Uma sessão pode ser atribuída a um canal por uma das seguintes três formas.

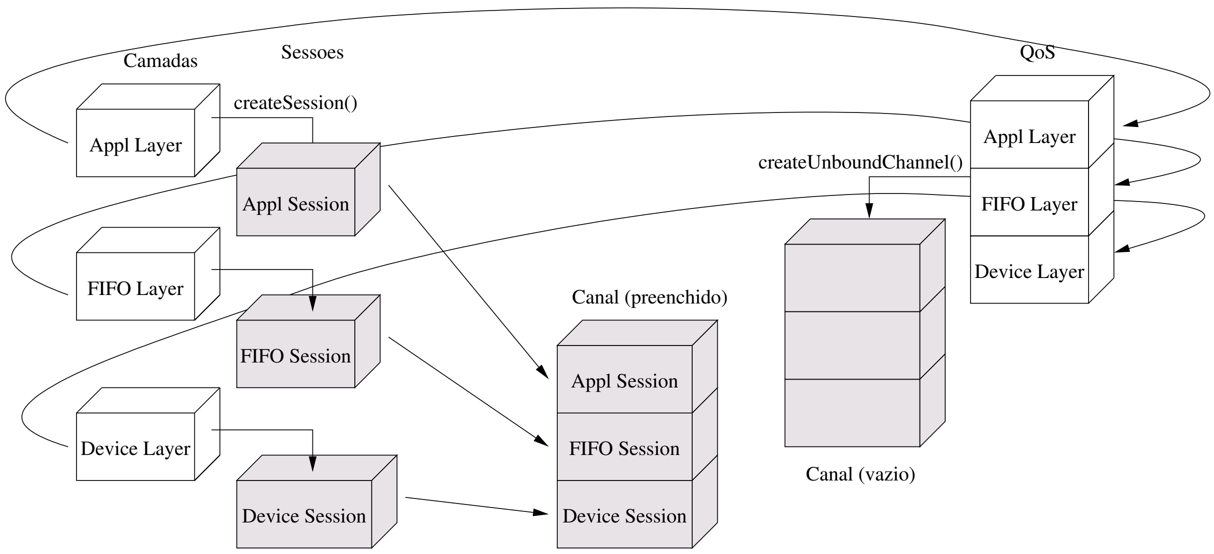

Figura 4.5: Instanciação de canais a partir de QoSs.

Explicitamente. A atribuição explícita permite à entidade que está a definir o canal nomear uma sessão, que pode participar já em outros canais.

A atribuição explícita oferece a possibilidade de definir configurações complexas. No exemplo dos grupos ligeiros, apresentado anteriormente, todos os canais devem utilizar a mesma sessão do protocolo LWG.

Automaticamente. A atribuição automática é realizada convidando as sessões que foram atribuídas explicitamente ao canal a participarem, elas próprias, no processo.

A atribuição automática permite aos protocolos que utilizam configurações especiais intervir no processo de definição do canal. No caso dos grupos ligeiros, a sessão comum do protocolo LWG pode definir as sessões que participarão no canal, optimizando os recursos consumidos.

Por omissão. Concluído o processo de atribuição automática, o *Appia* irá verificar se existem posições por preencher no vector. As posições livres serão preenchidas por sessões novas, solicitadas à camada que se encontra na posição correspondente da QoS que modelou o canal.

O processo de atribuição é executado pela ordem acima: após a atribuição explícita, o canal executa a atribuição automática. As posições não atribuídas por nenhum destes modos serão inicializadas por omissão. Os três modos de atribuição de sessões a canais estão representados na figura 4.6.

4.2.2 Composições não convencionais

Os modelos de atribuição enunciados são cruciais na flexibilidade que o *Appia* apresenta no suporte a composições complexas. Mantendo a abstracção de canal, o sistema impõe restrições significativamente menores à composição ao permitir que as sessões participem em mais de um canal.

O *Appia* praticamente não impõem restrições ao resultado final da composição, que pode aproximar-se de uma pilha, ou ser suficientemente aberto para ultrapassar os limites da composição em árvore. Por definição, os grafos resultantes das composições do *Appia* são acíclicos, visto que não é permitido a uma sessão ocorrer mais que uma vez na composição. A grande vantagem do *Appia* relativamente aos restantes sistemas analisados é o facto de aliar a ligação tardia das camadas a esta flexibilidade. O código

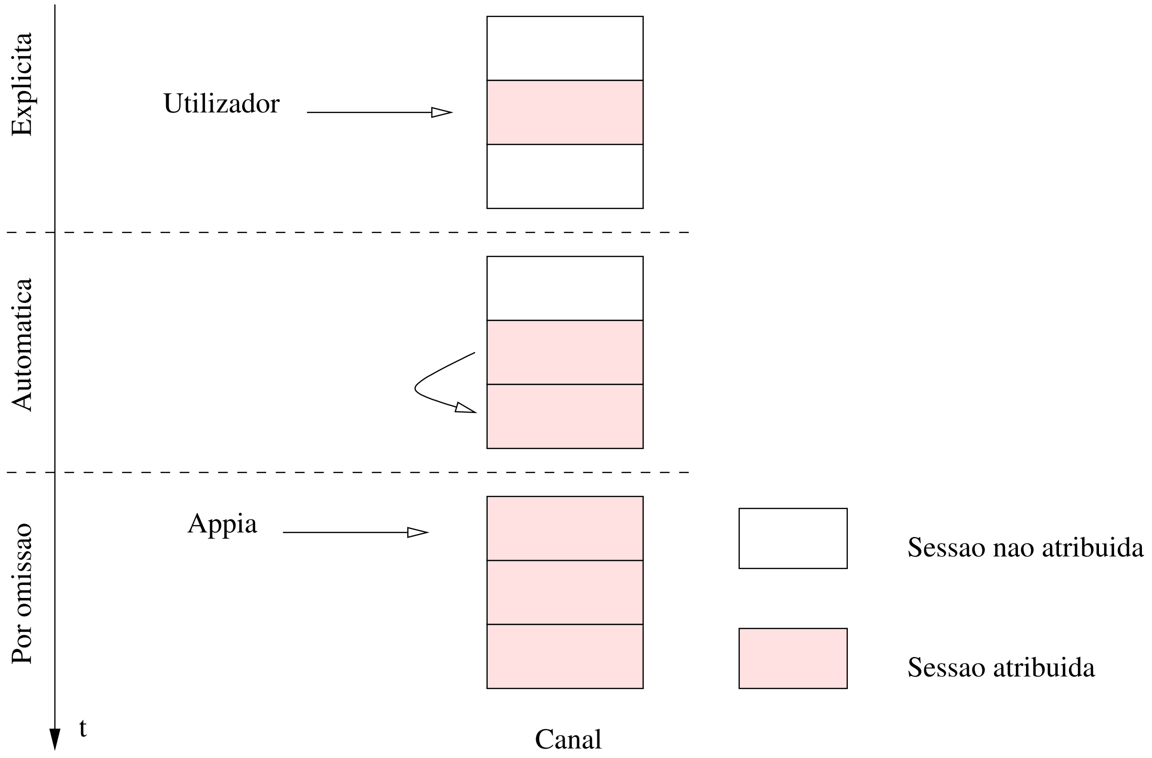

Figura 4.6: Exemplo de atribuição de sessões a um canal com três posições

que concretiza um protocolo pode manter-se válido, independentemente do número de canais em que cada sessão participa e da configuração que apresentem.

O modelo de composição do *Appia* satisfaz parcialmente a delimitação de nível de aplicação (apresentada na secção 2.3.5.1). A aplicação pode, em função da mensagem, determinar quais as propriedades que lhe deverão ser aplicadas indicando, por exemplo, quais delas deverão ser retransmitidas em caso de erro. Concretizações mais evoluídas dos protocolos, poderão também beneficiar das sugestões descritas na secção 2.3.5.2, beneficiando ainda de não caber necessariamente ao programador da aplicação a concretização de parte dos protocolos. Diferentes canais poderiam conter diferentes sub-protocolos, partilhando as propriedades nucleares do protocolo. Retomando o exemplo do TCP apresentado então, dois canais poderiam partilhar as propriedades fundamentais deste protocolo, codificadas como um protocolo que participaria em ambos. Um segundo protocolo complementaria o canal utilizado para as sessões remotas, introduzindo algum atraso no envio das confirmações.

4.2.3 Eventos

O *Appia* reune as vantagens de um modelo de eventos fechado com a flexibilidade oferecida pelo modelo aberto. Os eventos são objectos descendentes da classe EVENT. Esta classe tem três atributos: channel, que contém o canal onde o evento circula; direction, que indica o sentido em que o evento atravessa o canal e source, que refere a sessão que criou o evento.

Os protocolos podem criar novos eventos, fazendo as suas classes descender de EVENT ou de uma sua sub-classe. A herança desempenha um papel fundamental na reusabilidade dos protocolos: versões sucessivas de um mesmo protocolo podem enriquecer os eventos que definem com novos atributos, desde que estes descendam dos eventos criados pelas versões anteriores. As relações de dependência entre protocolos tornam-se mais fracas: assumindo que se mantém a semântica dos eventos, um protocolo pode ser composto com qualquer versão posterior de um outro, do qual depende.

A classe EVENT é extendida pela plataforma em duas direcções: eventos a enviar para a rede (classe SENDABLEEVENT) e eventos de interacção com o núcleo, como por exemplo, temporizadores, (classe CHANNELEVENT). Conceptualmente, o núcleo envolve o canal, recebendo todos os eventos que ultrapassem as camadas mais acima e mais abaixo da composição.

4.2.4 Validação da composição

No momento em que uma QoS é definida, as camadas declaram três conjuntos de eventos: Ψ_p contendo os eventos que o protocolo p necessita para uma execução correcta; Φ_p contendo os eventos que p está preparado para receber e Γ_p contendo os eventos que p irá introduzir no canal.

Para o *Appia*, uma composição correcta será aquela em que o conjunto dos eventos declarados como necessários são introduzidos no canal. Ou seja, uma composição é correcta quando $\Psi \subseteq \Gamma$, sendo Ψ (resp. Γ) a união de todos os Ψ_p (resp. Γ_p) da composição. As QoSs que não respeitem a restrição anterior não serão aceites pelo sistema.

A satisfação da restrição não assegura a validade da composição. Em particular, não é contemplada a semântica nem a direcção dos eventos. No entanto, das relações entre protocolos identificadas na secção 2.1, o *Appia* consegue assinalar parcialmente o incumprimento de relações de dependência.

O Ensemble adopta uma perspectiva mais forte da validação da composição ao verificar a correcção do protocolo. No entanto, este processo não pode ser realizado em tempo de execução, sendo necessário submeter previamente a composição a um ambiente próprio.

Os conjuntos Φ_p são utilizados em optimizações à execução. É esperado que todos os protocolos satisfaçam a condição $\Psi_p \subseteq \Phi_p$.

4.2.5 Encaminhamento de eventos

Alguns protocolos só se tornam activos quando ocorrem determinadas condições (Bhatti *et al.*, 1998; van Renesse *et al.*, 1996). Por exemplo, numa pilha que ofereça suporte à comunicação em grupo, a camada que concretiza o protocolo de instalação de vistas não adiciona nenhuma informação relevante às mensagens de dados trocadas pela aplicação. Retomando a definição de protocolo da equação 2.1 (na página 6), para alguns eventos, a função protocolo converte-se na função identidade, ou seja, $P(S, E) \to (S, \{E\})$.

Os conjuntos Φ_p declarados pelas camadas aquando da definição das QoSs são utilizados para criar caminhos optimizados de eventos. Para cada evento $e \in \Gamma$, a QoS gera uma tabela contendo a sequência das camadas que mencionaram e no seu conjunto Φ_p. Em tempo de criação de canal, o *Appia* define tabelas de encaminhamento de eventos projectando as camadas nas sessões que serão utilizadas. Sempre que um evento é introduzido no canal, o *Appia* utiliza a informação da tabela associada ao tipo de evento para determinar o conjunto de sessões que terá de visitar. Do ponto de vista da concretização, o caminho a percorrer por cada evento é mantido apenas à custa de um vector de sessões. De notar que o esforço computacional para a determinação das tabelas de encaminhamento de eventos está concentrado no momento da definição das QoSs, que se espera que ocorra no início da execução da aplicação. Desta forma, a eficiência da plataforma não é comprometida pela flexibilidade que apresenta.

No *Appia*, à semelhança do Ensemble (Hayden, 1998), a correspondência entre um evento e o canal em que circula é avaliada uma única vez, no momento em que é recebido no canal. O sistema utiliza por isso a composição mista, pormenorizada em 2.3.3. Contudo, o Ensemble utiliza funções do núcleo para receber as mensagens da rede e para determinar o canal em que circularão. O *Appia* apresenta um modelo mais flexível por delegar nos protocolos ambas as operações.

O mecanismo de herança é fundamental para o correcto funcionamento do sistema. Os testes aos tipos de eventos necessários para a concretização dos algoritmos têm sempre em consideração a herança apresentada pelos objectos. Evita-se assim que esta facilidade limite as possibilidades de adaptação dos protocolos aos ambientes em que são executados. O *Appia* extende o Coyote neste aspecto: neste, o conjunto de eventos utilizado na composição é fixo e conhecido à partida por todas as camadas, gerando indesejáveis relações de dependência entre os protocolos.

Um dos efeitos da definição dos trilhos de eventos no Ensemble é a supressão do código que não realiza sobre um evento qualquer operação útil. O *Appia* oferece um resultado semelhante como facilidade por omissão da aplicação, evitando a anotação do código e a preparação *à priori* da composição. O número de camadas que processa cada mensagem fica por isso dependente das suas características, gerando assim, composições minimais por evento.

4.2.5.1 Apresentação de mensagens a protocolos

O *Appia* dispõe de dois mecanismos que condicionam a apresentação de mensagens aos protocolos: a utilização de múltiplos canais e os caminhos de eventos. Embora seja possível encontrar intersecções nos resultados conseguidos com ambos, cada um deles tem um âmbito de aplicação distinto.

A utilização de diferentes canais apresenta-se como uma facilidade a ser utilizada pela aplicação ou por camadas especiais, com conhecimento sobre os canais em que estão integradas. O objectivo é proporcionar diferentes qualidades de serviço, com a possível partilha de propriedades. Efectuando uma projecção dos eventos que as camadas subscrevem em canais, é possível definir um conjunto de canais que oferece um serviço equivalente ao dos caminhos de eventos. No entanto, as camadas teriam forçosamente que ter conhecimento sobre a composição em que estavam inseridas, diminuindo as suas possibilidades de reutilização.

Os caminhos de eventos não são mais que optimizações, com impacto no código dos protocolos e no desempenho. A utilização de eventos para atribuição de diferentes propriedades implica concretizações especializadas dos protocolos ou incapazes de cooperar entre si, por receberem eventos cujo tipo é ignorado pelos restantes protocolos.

4.2.6 Configurabilidade

Uma característica importante dos protocolos no *Appia*, é a sua capacidade de reconhecer e actuar diferentemente, quando uma mesma sessão participa em diversos canais. No entanto, as possibilidades de reutilização dos protocolos seriam muito enfraquecidas se esta propriedade fosse obrigatória. Uma sessão de um protocolo de ordenação, por exemplo, não terá interesse em reconhecer a existência dos diversos canais, uma vez que a propriedade que ele providencia deverá ser aplicada a todas as mensagens. A camada de grupos ligeiros, pelo contrário, deverá actuar de forma diferenciada e reconhecer as propriedades que compõem cada um dos canais para assegurar que providencia as propriedades requeridas por todos os grupos.

A informação sobre o canal em que um evento circula é sempre apresentada às sessões mas a sua utilização é opcional. Para a recepção e reenvio de um evento, a sessão não necessita de consultar este valor. Para gerar novos eventos (por exemplo uma resposta), a sessão deve copiar para o novo evento o valor que recebeu, tratando-o como um valor opaco. Esta característica permite reduzir a dimensão dos cabeçalhos adicionados pelos protocolos às mensagens: a informação de endereçamento é dispensável para as camadas que não reconhecem os canais em que estão inseridas e opcional para as restantes. Diminui-se assim a dimensão das mensagens e, consequentemente, um dos factores prejudiciais ao desempenho enunciados em 2.3.5.

Embora diminuindo a reusabilidade, um protocolo pode utilizar o conhecimento sobre os canais em que está inserido para aumentar a sua configurabilidade. Uma sessão pode, por exemplo, avaliar as diferentes QoSs disponíveis, para seleccionar o canal adequado para o tipo de mensagens que pretende enviar. No caso do x-Kernel , não é possível ao programador determinar em tempo de execução as diferentes qualidades de serviço a que um protocolo está associado, em parte porque o seu modelo não antecipa essa necessidade. Neste sistema, a associação de uma qualidade de serviço a uma mensagem é da exclusiva responsabilidade do programador das aplicações.

4.2.7 Concorrência

O *Appia* não impõe ao programador nenhum dos modelos de concorrência enumerados na secção 2.3.2, permitindo assim que o sistema se adeqúe aos ambientes em que será executado. São oferecidas ao programador as duas garantias comuns em praticamente todos os outros sistemas: a manutenção da ordem dos eventos dentro do canal e a impossibilidade de um protocolo executar simultaneamente dois eventos.

No entanto, dadas as características do sistema, nomeadamente, a participação das sessões em múltiplos canais e os caminhos de eventos, assegurar estas propriedades requer alguns cuidados adicionais.

4.3 Desenho

A figura 4.7 apresenta o modelo UML[2] do *Appia* (Miranda *et al.*, 2000b). A concretização respeitou, de forma geral, o modelo apresentado. O desenho assenta sobre cinco classes nucleares: LAYER, SESSION, QOS, CHANNEL e EVENT que correspondem, respectivamente, aos conceitos de camada, sessão, QoS, canal e evento, já apresentados.

[2]Unified Modeling Language

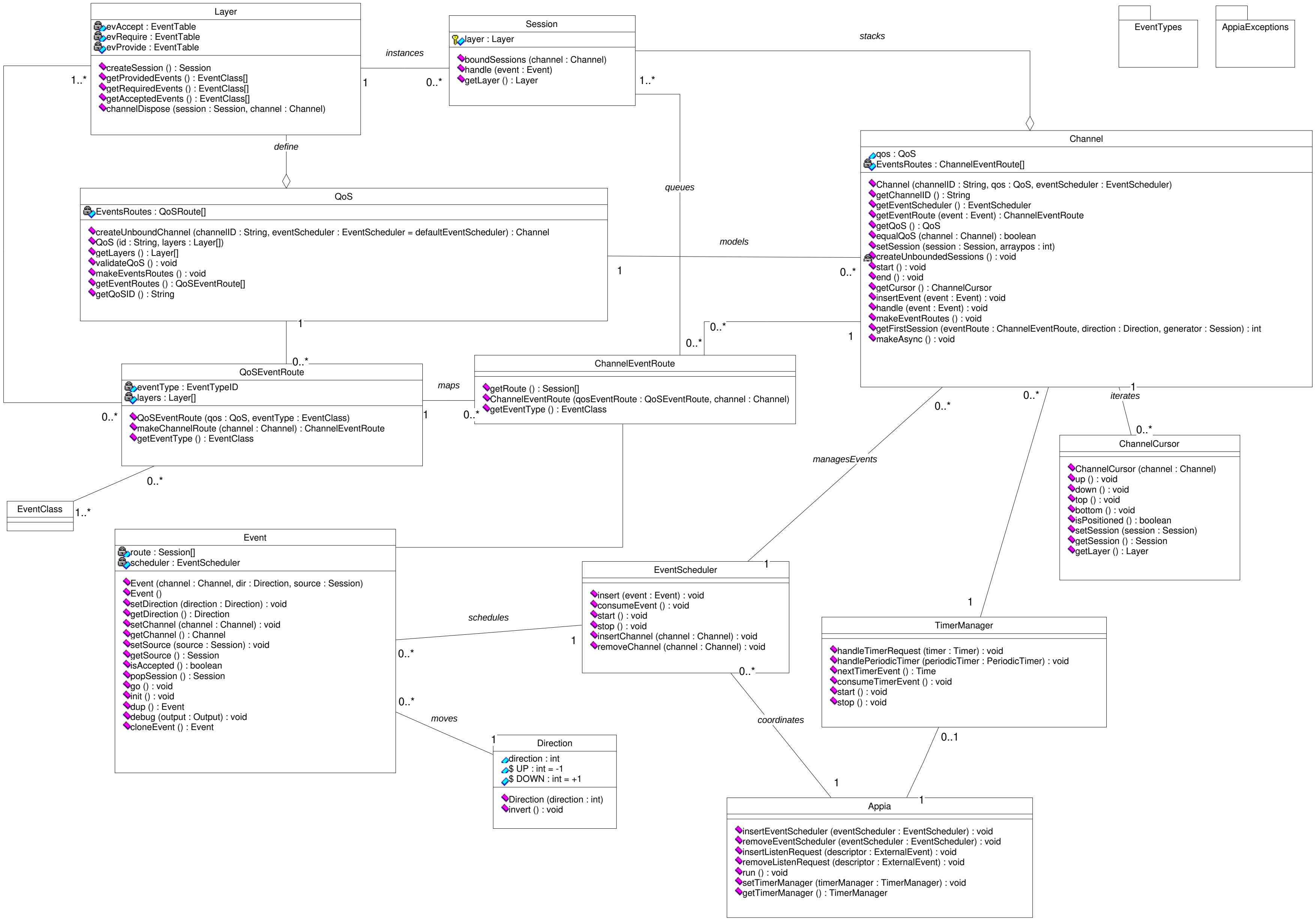

Figura 4.7: Modelo UML do *Appia*

4.3.1 Construção de QoSs e canais

Uma QoS é definida por um vector de objectos da classe abstracta LAYER (figura 4.8). Esta classe será especializada pelos programadores dos protocolos que terão, necessariamente, que especificar os três conjuntos de eventos referidos na secção 4.2.4: no vector evProvide o conjunto Γ_p, em evRequire o conjunto Ψ_p e em evAccept o conjunto Φ_p. No momento da criação do objecto, o construtor da classe QOS invocará os métodos getProvidedEvents, getRequiredEvents e getAcceptedEvents de todos os objectos do tipo LAYER do vector e que devolvem os vectores indicados. Os dois primeiros métodos são usados na validação da composição e o último na criação dos caminhos de eventos.

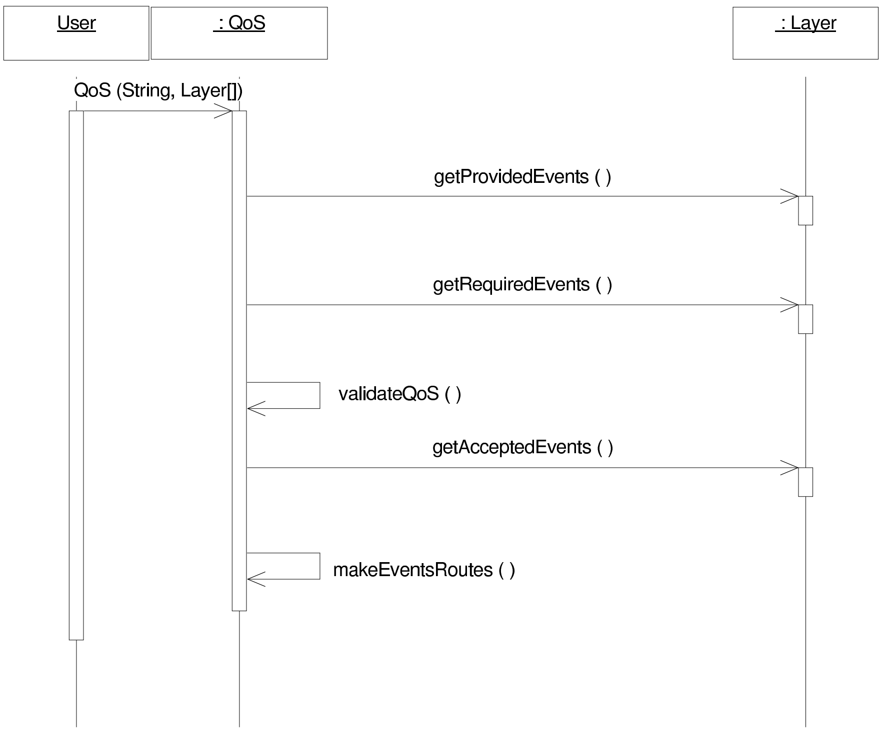

Figura 4.8: Diagrama de interacções da definição de uma QoS

O código que concretiza cada um dos protocolos será definido em sub-classes de SESSION. Um canal é criado invocando o método createUnboundChannel da QoS. As atribuições explícita e automática de sessões a canais são realizadas utilizando objectos da classe CHANNELCURSOR, solicitados ao canal. O cursor desloca-se sobre as posições do canal e valida as atribuições realizadas, verificando se a sessão que está a ser atribuída ao canal foi criada pela camada correspondente na QoS. A figura 4.9 apresenta o diagrama de interacções em que, durante o processo de atribuição explícito ou automático, é solicitada a uma LAYER que gere uma nova SESSION, a qual é atribuída ao canal em definição.

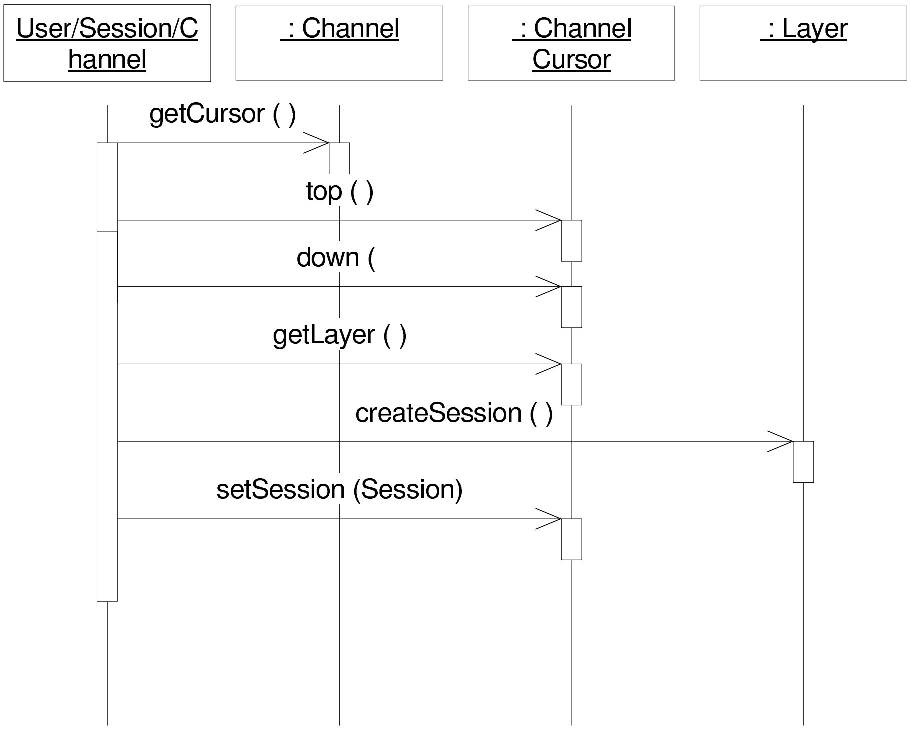

Figura 4.9: Diagrama de interacções da atribuição de uma sessão a um canal

A atribuição explícita é realizada no momento em que o canal é criado, antes da invocação do método start do canal, o qual coordenará as atribuições automáticas e por omissão. A figura 4.10 especifica os procedimentos executados pelo método start: invocação do método boundSessions das sessões já atribuídas, procedendo assim à atribuição automática, e do método createUnboundedSessions que tem a responsabilidade de completar a atribuição de sessões a canais.

As classes QOSEVENTROUTE e CHANNELEVENTROUTE colaboram na definição dos caminhos de eventos. Para cada evento que circulará num canal, ou seja, para cada evento e declarado num dos vectores evProvide das camadas que integram a qualidade de serviço, a classe QOS cria um objecto da classe QOSEVENTROUTE, contendo referências para as camadas que o mencionaram (ou a uma classe de eventos da qual

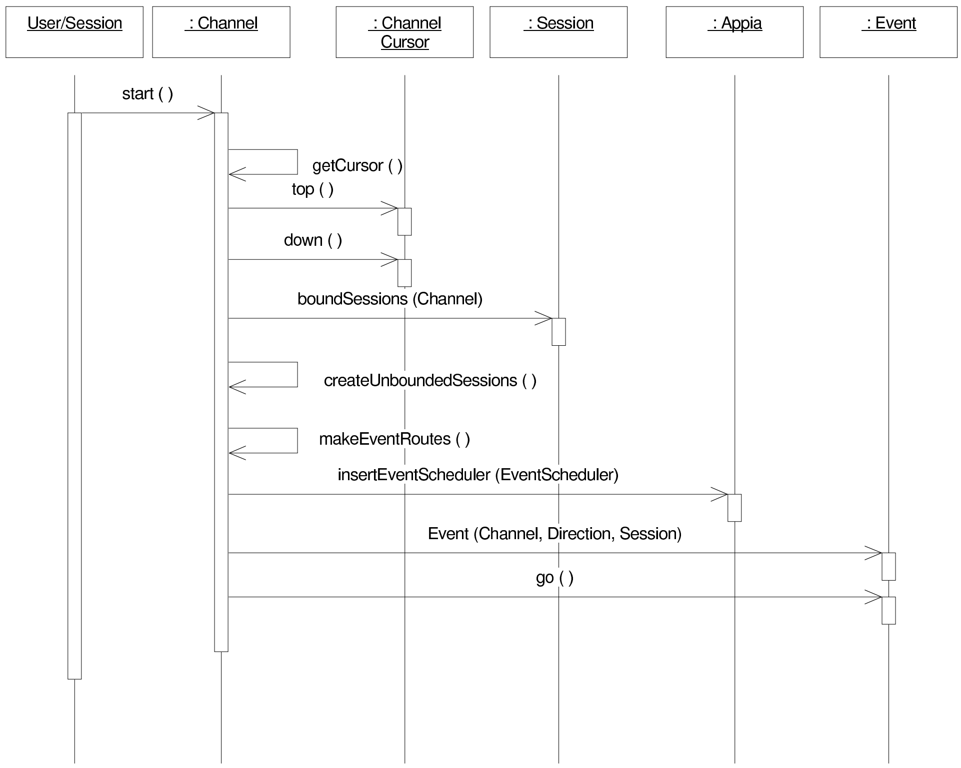

Figura 4.10: Sequência de operações realizadas pelo método start da classe CHANNEL

e descenda) no vector evAccept. Em tempo de construção do canal, e após todas as posições do canal terem sido atribuídas, cada instância de QOSEVENTROUTE servirá como base à criação de um objecto do tipo CHANNELEVENTROUTE, pela projecção das camadas que participam na QoS, nas sessões que participarão no canal.

O processo de criação de um canal termina com o registo do escalonador de eventos do canal no sistema e a introdução de um evento do tipo CHANNELINIT.

4.3.2 Eventos

Para que um evento possa circular no canal têm que ser definidos os três atributos necessários para o seu encaminhamento: canal (atributo channel), direcção (atributo direction) em que circula e sessão (atributo source) que o definiu. O tipo do evento e o atributo channel são utilizados na determinação da instância de CHANNELEVENTROUTE adequada. O atributo source posiciona o evento no CHANNELEVENTROUTE

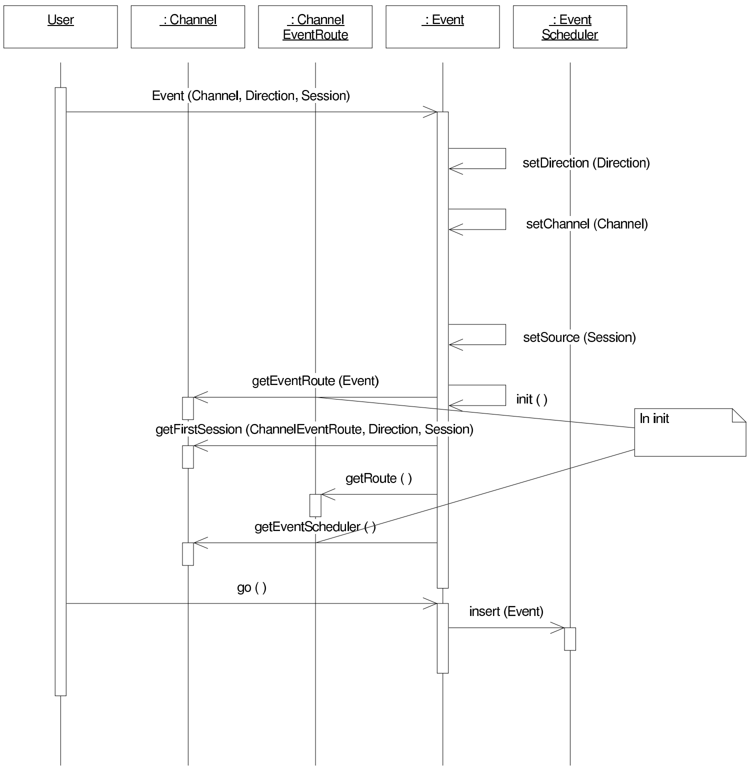

Figura 4.11: Sequência de operações realizadas pelo construtor da classe EVENT

enquanto direction indica a direcção em que o evento percorrerá o vector de sessões contido no CHANNELEVENTROUTE.

A classe EVENT tem dois construtores: um construtor vazio e um em que todos os atributos são passados como argumento. No caso do primeiro, os atributos podem ser definidos individualmente invocando os métodos setChannel, setSource e setDirection. A inicialização do evento, nomeadamente a localização do CHANNELEVENTROUTE e o posicionamento do evento no caminho, são realizados pelo método init, implicitamente invocado pelo construtor que aceita todos os atributos como argumentos. A figura 4.11 apresenta o processo de criação de um evento utilizando o construtor que recebe argumentos.

Na figura 4.12 é apresentado o conjunto base de eventos da plataforma. Para reduzir os condicionamentos impostos aos protocolos a desenvolver, o modelo é intencionalmente limitado às duas categorias anteriormente referidas: eventos para comunicação com o canal (família CHANNELEVENT) e eventos a transmitir pela rede (família SENDABLEEVENT).

Os eventos que circularão pela rede foram normalizados pela plataforma por duas razões. Em primeiro lugar porque a plataforma é intencionalmente orientada à comunicação entre processos. A inexistência de uma normalização, mesmo que mínima, sobre esta classe de eventos levaria a que fossem definidos protocolos incompatíveis entre si. A segunda porque se torna mais simples integrar uma biblioteca optimizada de operações usuais sobre mensagens.

Na perspectiva do sistema, as sessões são objectos passivos: apenas reagem à entrega de eventos. O *Appia* não oferece por isso, qualquer mecanismo de controlo de concorrência que o defenda da geração "espontânea" de eventos pelas sessões, à excepção do método async da classe CHANNEL, apresentado em seguida.

4.3.2.1 Eventos do canal

Todos os eventos da família CHANNELEVENT têm um atributo do tipo EVENTQUALIFIER. As instâncias desta classe são caracterizadas por um de três valores: ON, OFF ou NOTIFY. O significado de cada uma destas constantes é particular ao tipo do evento.

O canal disponibiliza temporizadores periódicos e aperiódicos, identificados respectivamente pelas classes TIMER e PERIODICTIMER. No caso dos eventos aperiódicos, a instância do objecto utilizada na solicitação será também utilizada na notificação. O qualificador de evento será usado para distinguir a ocorrência, contendo respectivamente as constantes ON e NOTIFY. A constante OFF é utilizada pela camada para solicitar o cancelamento do temporizador. A reutilização de instâncias de objectos apresenta vantagens no desempenho e na simplicidade de concretização dos protocolos. No desempenho por dispensar uma execução dos processos de reserva e libertação de memória. Na concretização por permitir a utilização da referência para o objecto como identificador. No entanto, a classe suporta também a utilização dos identificadores convencionais, definidos por cadeias de caracteres. As notificações de eventos periódicos são realizadas por cópias do evento original.

Quando é introduzido no canal, um ECHOEVENT transporta como atributo uma

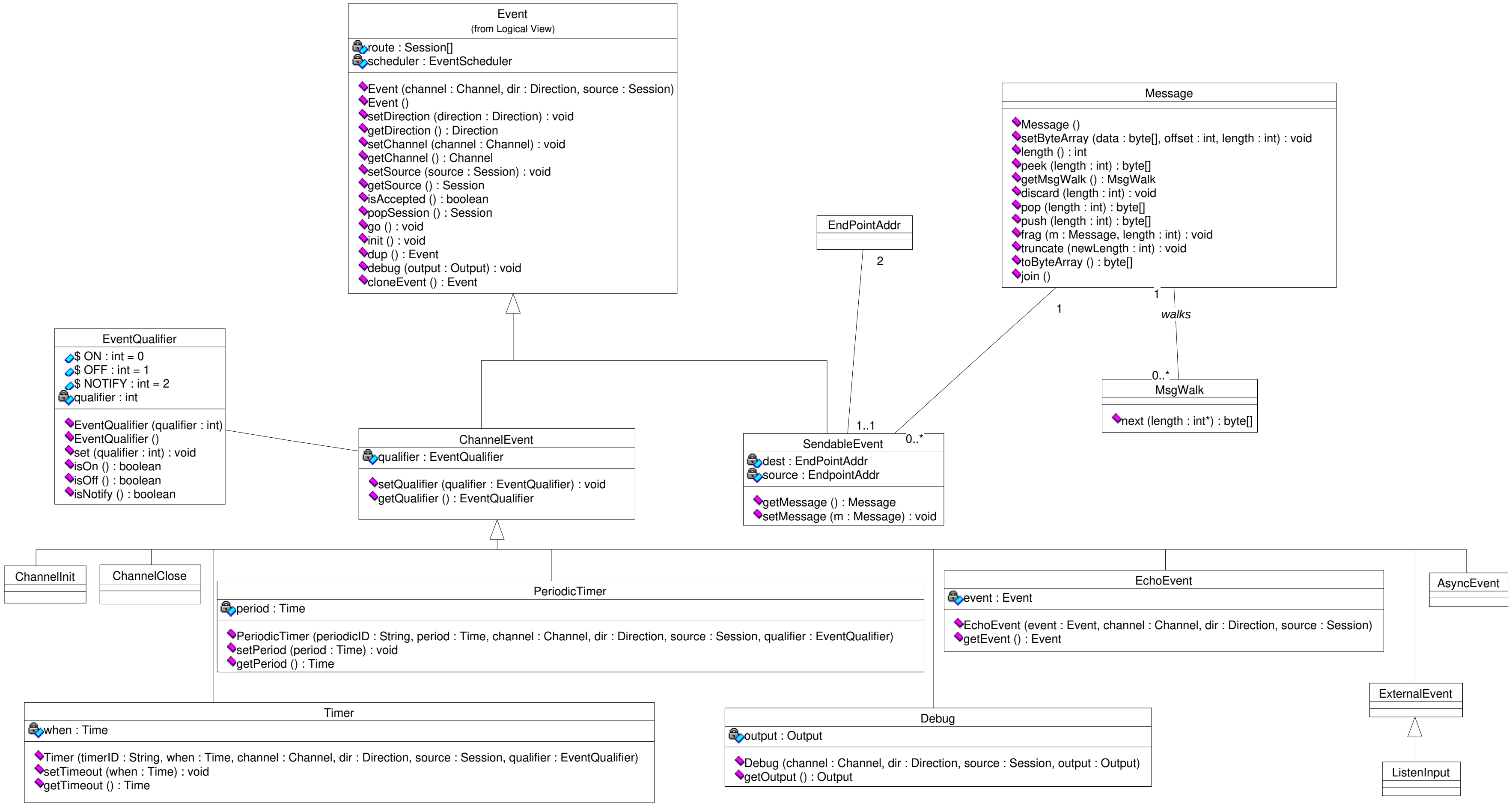

Figura 4.12: Diagrama UML de eventos do *Appia*

instância de um outro evento. Ao atingir uma das margens da pilha, o sistema extrai o evento transportado e insere-o no canal, no sentido oposto aquele em que o ECHOEVENT viajou. Idealmente, este evento recolherá informação das sessões que atravessar até chegar novamente à sessão que gerou o ECHOEVENT. Um protocolo que realize fragmentação de mensagens, por exemplo, poderá utilizar os ECHOEVENT para recolher informação sobre a dimensão máxima dos fragmentos.

A classe de eventos ASYNC não pode ser extendida. Um evento da classe ASYNC é introduzido no canal cada vez que uma sessão invoca o método async do canal. Em toda a concretização do *Appia*, este método é o único cuja especificação assegura o controlo de concorrência. Estes eventos não apresentam qualquer outro atributo para além do conjunto herdado de CHANNELEVENT. A sua única missão é assegurar que a sessão que solicitou o evento terá brevemente que o tratar, conquistando assim o acesso à actividade que lhe permitirá introduzir eventos no canal.

A classe de eventos DEBUG normaliza o processo de depuração do código dos protocolos. Consoante o valor do qualificador, as sessões são convidadas a relatar o seu estado instantâneo (constante NOTIFY) ou a relatar os eventos que recebem e as respectivas alterações ao estado. Neste último caso, a constante ON activa o modo de depuração e a constante OFF desactiva-o.

4.3.2.2 Eventos contendo mensagens

A classe SENDABLEEVENT normaliza os eventos que contêm mensagens a ser enviadas pela rede. A classe apresenta apenas o conjunto mínimo de atributos necessários para esta operação: origem, destino e mensagem.

O *Appia* não define o tipo dos atributos source e dest. Cabe aos protocolos envolvidos acordarem um formato, que pode mesmo ser alterado ao longo do percurso do evento pelo canal. Por exemplo, numa composição que suporte comunicação em grupo, uma camada pode substituir o nome do grupo de destino pelo endereço de difusão IP respectivo.

A classe MESSAGE partilha o mesmo objectivo do conjunto de funções com o mesmo nome do x-Kernel (Mosberger, 1996): aumentar o desempenho da plataforma pela criação de uma biblioteca de operações comuns sobre mensagens que reduzam o número de operações de cópia. Esta classe assume que as mensagens são criadas usando um modelo de pilha: antes do envio, cada sessão prefixa a mensagem com o seu cabeçalho e na recepção, cada sessão retira a mesma quantidade de octetos, também do início da mensagem.

4.3.3 Execução

4.3.3.1 Escalonamento de eventos e controlo de concorrência

Os escalonadores de eventos, representados no *Appia* pela classe EVENTSCHEDULER, têm a responsabilidade de manter a ordenação FIFO e entregar às sessões os eventos de um ou mais canais, invocando o respectivo método handle. A interface desta classe foi desenhada para permitir a aplicação de diversas políticas de escalonamento. O âmbito das opções cobre os critérios de ordenação de eventos e a utilização de actividades. Extensões à concretização actual desta classe podem assim contemplar a existência de prioridades de eventos e processamento concorrente.

Para facilitar o controlo de concorrência, o escalonamento de eventos é concretizado em dois níveis. Em tempo de criação do canal, é possível indicar a instância da classe EVENTSCHEDULER a utilizar. Por omissão, cada canal utiliza um escalonador de eventos próprio, definindo relações de "um para um" com o canal. O segundo nível é concretizado pela classe APPIA, que actua como "meta-escalonador" ao invocar ciclicamente o método consumeEvent de cada escalonador de eventos registado. O processo de tratamento de um evento está representado na figura 4.13. No *Appia*, a informação necessária ao seu encaminhamento está encapsulada no próprio evento. É ele quem informa o escalonador sobre a próxima sessão a visitar e se re-insere no escalonador quando o método go é invocado. As sessões interagem exclusivamente com os eventos. O escalonador de eventos é-lhes totalmente transparente.

As sessões são a unidade de execução dos protocolos no *Appia*, interagindo com o ambiente através da recepção e envio de eventos. Estes são entregues às sessões

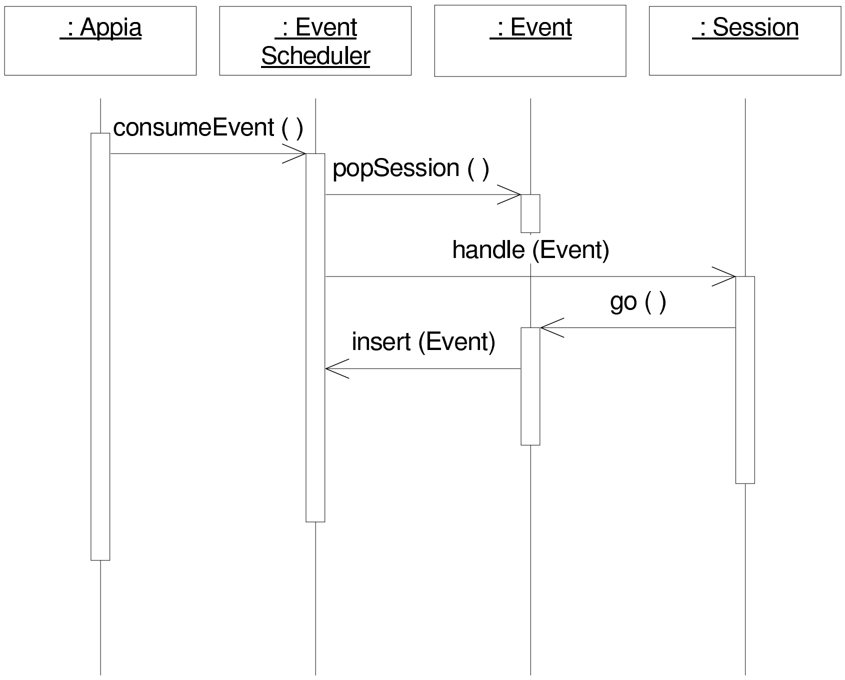

Figura 4.13: Sequência de operações realizadas para o processamento de um evento por uma sessão

pela invocação do método handle, que se torna assim o ponto central da codificação de qualquer protocolo. A distinção entre os diferentes tipos de eventos será realizada pelo protocolo utilizando operadores da linguagem de programação, enquanto que o estado dos atributos fundamentais do evento poderá ser obtido invocando os métodos getSession, getChannel e getDirection.

4.3.4 Herança

A herança assume um papel importante na versatilidade e desempenho da plataforma, quer na concretização dos protocolos quer na utilização dos eventos.

4.3.4.1 Protocolos

As duas componentes em que o *Appia* divide os protocolos, nomeadamente a camada e a sessão são concretizadas como sub-classes, respectivamente de LAYER e

SESSION. O esforço de concretização de um protocolo pode ser decomposto em duas componentes: o algoritmo e a sua integração na plataforma. A reusabilidade do código, característica do mecanismo de herança, reduz substancialmente o esforço de concretização da segunda.

4.3.4.2 Eventos

Os eventos no *Appia* beneficiam da herança ao fomentar a reusabilidade dos protocolos e, quando visto numa perspectiva que integra também os conjuntos de eventos declarados pelas camadas, ao melhorar o desempenho da plataforma.

À semelhança do Ensemble, o *Appia* utiliza um encapsulamento forte dos protocolos. Na secção 3.5.4 foi referido que a utilização de eventos para a solicitação e notificação de temporizadores prejudicava o desempenho da plataforma, por implicar um consumo adicional de recursos associado ao processamento dos eventos por todas as camadas. A resolução deste problema no *Appia* passa por utilizar a herança para que cada protocolo especialize as classes TIMER e PERIODICTIMER. Esta especialização apresenta duas vantagens de desempenho, apresentadas em seguida.

Desempenho do canal O desempenho do canal fica beneficiado se cada protocolo declarar na sua camada que pretende receber apenas os temporizadores declarados por si, ou seja, aqueles cujo tipo corresponde à especialização realizada e não ao tipo base TIMER ou PERIODICTIMER. Desta forma, e uma vez que na construção das tabelas de encaminhamento de eventos é também utilizada a herança, um temporizador visitará apenas a sessão de destino, evitando chamadas desnecessárias às outras sessões.

Desempenho da sessão A utilização de especialização é transparente para a plataforma. Tipicamente, um pedido de temporizador tem associado alguma informação específica. Por exemplo, um protocolo que assegure a fiabilidade da entrega poderá solicitar um temporizador por cada mensagem enviada, procedendo à sua retransmissão se até ao momento em que recebe a notificação não tiver recebido uma confirmação. No *Appia*, a informação de contexto pode fazer parte do evento que solicitou o temporizador, dispensando o tempo necessário à localização da informação de contexto a partir de um identificador. O protocolo FIFO, concretizado para a plataforma, inclui uma referência para a mensagem transmitida nas instâncias do temporizador que solicita para possível retransmissão.

4.4 Concretização

O *Appia* e um conjunto base de protocolos de entrega ponto-a-ponto com ordenação FIFO, fragmentação e suporte à comunicação em grupo estão concretizados na linguagem Java e disponíveis no servidor do protocolo HTTP acessível em `http://appia.di.fc.ul.pt`.

A escolha da linguagem Java para a concretização é fundamentada na orientação aos objectos e portabilidade que apresenta e pela aceitação que tem tido junto da comunidade científica e de produção. Facilidades adicionais, como a reciclagem automática de memória vieram a mostrar-se muito vantajosas durante o desenvolvimento. Por outro lado, foi sentido que algumas das características de alto nível apresentadas pela linguagem impedem a concretização de optimizações correntes em outras linguagens, como o C. A face mais visível é a impossibilidade de aceder directamente a endereços de memória.

Nos próximos parágrafos serão discutidas algumas opções relacionadas com a concretização do desenho apresentado anteriormente e a forma como o Java beneficiou ou prejudicou o esforço de desenvolvimento.

4.4.1 Mensagens

A classe MESSAGE apresenta uma interface com operações que permitem manipular as mensagens como pilhas, às quais são adicionados ou retirados conjuntos de octetos. Durante a construção de uma mensagem (usualmente quando o evento que a transporta circula no sentido descendente), a memória é reservada em blocos de 1000 octetos. Cada um destes blocos será preenchido com qualquer número de cabeçalhos, desde que não ultrapasse o espaço disponível no bloco. Sempre que tal aconteça, o objecto reserva um novo bloco, que adiciona ao início da lista de blocos da mensagem. As funcionalidades da classe MESSAGE foram inspiradas na interface homónima desenvolvida para o x-Kernel (Mosberger, 1996).

O Java apresenta limitações que impediram uma concretização directa deste modelo. Ao contrário do que acontece no x-Kernel , não foi possível criar operações de *push* e *pop* (respectivamente de adição e remoção de um cabeçalho à mensagem) que devolvessem o primeiro endereço de memória a utilizar. Para conseguir um resultado equivalente, foi necessário adicionar ao modelo a classe MSGBUFFER, que define um apontador para o interior de uma mensagem através de uma referência para um bloco e de um deslocamento para o seu interior. As operações push e pop devolvem uma referência para um destes objectos, indicando a posição onde o cabeçalho deverá ser colocado ou se encontra. Esta solução apresenta ainda algumas desvantagens: a criação e a libertação de objectos da classe MSGBUFFER introduzem atraso no processamento. O problema pode ser atenuado reutilizando as instâncias da classe. Este mecanismo não é intuitivo para os programadores de Java e, portanto, aumenta a complexidade do desenvolvimento dos protocolos.

4.4.1.1 Conversão de objectos a octetos

O *Appia* não utiliza as facilidades de conversão de objectos para conjuntos de octetos do Java. Seguindo o modelo clássico, cabe às sessões a responsabilidade de converter os dados em vectores de octetos e, na recepção, de realizar a operação inversa. Os protocolos que fazem a ligação entre a plataforma e a rede (um utilizando o protocolo UDP e o outro o protocolo TCP) enviam para a rede apenas o conteúdo do atributo message, dos objectos que extendem a classe SENDABLEEVENT.

A utilização das facilidades do Java foi evitada por duas razões, descritas em seguida.

Portabilidade O Java converte os objectos num formato não normalizado, que dificilmente será reproduzido em outras linguagens de programação. A comunicação entre plataformas codificadas em diferentes linguagens de programação (por exemplo, C++) seria dificultada.

Conversão excessiva Os procedimentos de conversão definidos na linguagem incluem o objecto e todas as suas referências no vector de octetos resultante. Embora encapsulado para o programador, um evento contém referências para o canal e para o escalonador de eventos. Aplicar os procedimentos por omissão levaria à transmissão de dados irrelevantes para o destinatário. Uma primeira consequência seria o aumento do tempo de transmissão das mensagens.

4.4.2 Actividades

A concretização actual do *Appia* não trata concorrentemente eventos. No entanto, tal como referido na secção 4.3.3.1 o desenho do sistema permite que essa opção venha a ser concretizada. Concorrentemente com a actividade principal, o núcleo do *Appia* utiliza um conjunto adicional, dedicado a tarefas específicas, por exemplo, a gestão de temporizadores. Este modelo é semelhante ao utilizado por omissão no Ensemble (Hayden, 1998).

As características da linguagem não permitem a concretização da plataforma utilizando um único fluxo de controlo. A razão mais evidente é a inexistência de uma função com funcionalidades equivalentes à chamada `SELECT` da linguagem C. Ou seja, uma chamada que permita suspender um processo por um período máximo de tempo, ou até que sejam recebidos dados em um de um conjunto de descritores de ficheiros. Sem a utilização de actividades não é possível coordenar simultaneamente os pedidos de temporizadores das sessões enquanto se aguarda a recepção de dados, por exemplo, da rede.

As sessões são livres de recorrer a actividades para concretizar as suas funcionalidades. No entanto, cabe-lhes a responsabilidade de assegurar o controlo de concorrência nas interacções que realizam com os canais a que pertencem. Isto implica:

- assegurar que as interacções com o canal são realizadas apenas quando a sessão detem a actividade principal do *Appia*. Ou seja, quando o seu método handle foi invocado pelo núcleo do sistema. A excepção é o método async, seguro para execuções concorrentes;

- assegurar que invocações ao método handle nunca bloqueiam o processo;
- Sabendo que o *Appia* assegura que não são realizadas invocações concorrentes ao método handle, prevenir que actividades internas à sessão os provoquem, ou introduzir ferramentas de controlo de concorrência adequadas.

Para assegurar a máxima independência entre os protocolos e o sistema, cabe aos primeiros a responsabilidade de trocar dados com as entidades externas ao *Appia*. É o caso dos protocolos que interagem com a rede e com o utilizador. A solução que tem sido adoptada nos protocolos já concretizados, utiliza uma actividade da sessão para efectuar chamadas bloqueantes às entidades externas. Quando a actividade se desbloqueia (por exemplo, na recepção de um datagrama) invoca o método async do canal. Este, por sua vez, introduz um evento do tipo ASYNC o que, proximamente, entregará a actividade principal do sistema à sessão. Os dados recebidos podem assim ser inseridos no canal sem violar os requisitos da especificação.

4.4.3 Excepções

O conjunto de excepções do *Appia* é apresentado na figura 4.14.

As excepções foram agrupadas em categorias, associadas a classes. Sempre que uma categoria comporta diferentes conjuntos de ocorrências, como no caso da utilização do cursor de canal e dos eventos, o objecto é qualificado por uma constante, que identifica univocamente o erro ocorrido.

Do conjunto de situações que podem gerar um erro, merece destaque aquela que, dentro da categoria de excepções levantadas na utilização de eventos (classe APPIA-EVENTEXCEPTION), é identificada pela constante UNWANTEDEVENT. Esta excepção sinaliza ao criador de um evento que nenhuma das sessões do canal irá receber o evento introduzido no canal, ou seja, nenhuma das camadas que participou na definição

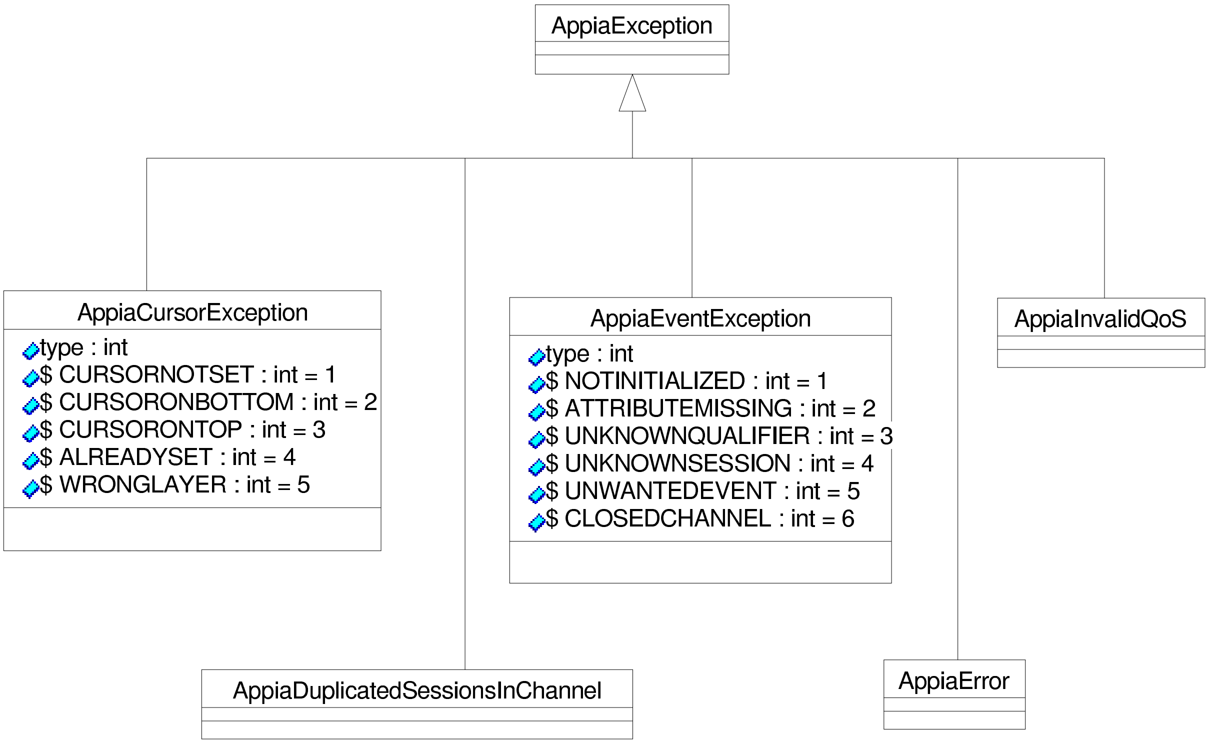

Figura 4.14: Diagrama de classes das excepções

da QoS declarou o evento no conjunto evAccept. Em termos de desempenho, esta informação poderá ser útil por permitir omitir a execução de partes do código do protocolo sempre que o canal não beneficie dessa informação.

4.5 Sumário

A primeira parte do capítulo apresenta dois exemplos, com aplicação prática, de composições irregulares de protocolos e discute a incapacidade da generalidade dos sistemas estudados em resolver, de forma elegante, o enunciado dos problemas.

Os requisitos necessários para a resolução de ambos os exemplos são enunciados e utilizados como motivação para a necessidade de um sistema que ofereça modelos de composição mais flexíveis.

Na segunda parte do capítulo é apresentado um novo sistema de suporte à composição de protocolos designado *Appia*, que extende os sistemas anteriores em diversas vertentes:

- oferece um modelo de composição mais flexível;
- permite aos programadores definirem protocolos reutilizáveis em contextos para os quais não foram inicialmente concebidos;
- introduz mecanismos inovadores de melhoramento do desempenho;
- facilita a criação de novas versões de protocolos que se mantêm compatíveis com composições anteriores;
- informa o programador de aplicações sobre a validade das composições realizadas;
- suporta a ligação tardia, tornando-se por isso adaptável, em tempo de execução, ao ambiente em que opera;
- mantém o encapsulamento dos protocolos;
- é facilmente configurável e adaptável a novos requisitos;

O sistema foi concretizado na linguagem Java, opção que o beneficia por se tornar utilizável, sem esforço significativo, em qualquer sistema computacional actual.

Notas

A necessidade de aumentar a flexibilidade dos sistemas de suporte à composição foi apresentada pelo autor em dois contextos diferentes, a que correspondeu a publicação de dois artigos. Em (Miranda & Rodrigues, 1999a) o problema é apresentado na sua vertente mais teórica, observando-se nas actas do Workshop em Groupware (Miranda & Rodrigues, 1999b) um trabalho especificamente orientada aos problemas das plataformas de trabalho cooperativo e multimédia.

A estrutura integral do *Appia* foi aceite para publicação na secção de cartazes das actas da 21ª Conferencia Internacional de Sistemas Computacionais Distribuídos (ICDCS'21) a realizar em Abril de 2001 (Miranda *et al.*, 2001).

5 Avaliação

A concretização do *Appia* é, neste capítulo, avaliada em duas vertentes. A primeira valida o sistema na concretização de soluções para as composições complexas apresentadas na secção 4.1. Na segunda é avaliado o desempenho do sistema. O capítulo é concluído com a apresentação de projectos onde a utilização do *Appia* se encontra presentemente a ser equacionada.

5.1 Concretização de protocolos

A utilização do *Appia* requerer dois tipos de intervenção, em momentos distintos: codificação e composição dos protocolos.

5.1.1 Codificação de protocolos

A figura 5.1 apresenta o código da classe FIFOLAYER que descreve os conjuntos de eventos recebidos, gerados e necessários ao correcto funcionamento de um protocolo de entrega ordenada e fiável de mensagens.

A cada SENDABLEEVENT no sentido descendente, a camada adiciona um número de sequência, que cresce linearmente com o número de mensagens trocadas com o destinatário.

Simultaneamente, o protocolo solicita ao núcleo da plataforma um temporizador, que sinalizará à camada a possível necessidade de realizar uma retransmissão. Este

```
public class FifoLayer extends Layer {

    /**
     * Usual Layer empty constructor
     */
    public FifoLayer() {
        super();

        /* Events enumeration */

        try {
            evProvide=new Class[4];
            evProvide[0]=Class.forName("appia.protocols.fifo.FIFOTimer");
            evProvide[1]=Class.forName("appia.protocols.fifo.AckEvent");
            evProvide[2]=Class.forName("appia.protocols.fifo.SendAckTimer");
            evProvide[3]=Class.forName("appia.protocols.fifo.FIFOUndeliveredEvent");

            evRequire=new Class[0];

            evAccept=new Class[7];
            evAccept[0]=Class.forName("appia.events.SendableEvent");
            evAccept[1]=Class.forName("appia.events.channel.ChannelInit");
            evAccept[2]=Class.forName("appia.protocols.frag.MaxPDUSizeEvent");
            evAccept[3]=Class.forName("appia.protocols.fifo.FIFOTimer");
            evAccept[4]=Class.forName("appia.events.channel.Debug");
            evAccept[5]=Class.forName("appia.protocols.fifo.SendAckTimer");
            evAccept[6]=Class.forName("appia.protocols.fifo.FIFOConfigEvent");
        }
        catch(ClassNotFoundException ex) {
            System.err.println("Class Not Found Exception in FifoLayer. "+
                        "Check CLASSPATH information.");
        }
    }

    /**
     * Standard session instantiation
     */
    public Session createSession() {
        return new FifoSession(this);
    }
}
```

Figura 5.1: Concretização da classe FifoLayer temporizador pertence a uma classe definida pelo protocolo e que extende a classe base TIMER.

A utilização de uma classe própria de temporizadores oferece duas vantagens com impacto na concretização: o programador recebe apenas os temporizadores que solicitou, eliminando-se a necessidade de verificar se o evento era efectivamente endereçado à camada; e a classe enriquece os atributos iniciais do temporizador, contendo em particular, o evento a ser retransmitido. Desta forma, é evitada uma operação de pesquisa em listas. Mais à frente neste capítulo demonstra-se que esta simplificação da concretização revela resultados positivos no desempenho.

O método handle da classe FifoSession é apresentado na figura 5.2. Em todos os protocolos concretizados, a estrutura deste método tem-se mostrado semelhante: uma sequência de testes ao tipo de evento recebido. De acordo com o resultado obtido, o programador encaminha o evento para um método especializado. Esta estrutura é semelhante à encontrada na concretização dos protocolos em outros sistemas, por exemplo, o Ensemble (Hayden, 1998). A estrutura do código não é, portanto, prejudicada pela interface do *Appia*.

A figura 5.3 apresenta o método da classe FIFOSESSION que procede à retransmissão das mensagens não confirmadas.

Entre as linhas 5 e 15 procede-se ao reenvio da mensagem. A mensagem, encapsulada dentro de um evento da classe SENDABLEEVENT, é obtida a partir dos atributos do temporizador, passado como argumento para o método (linha 6). A operação de cópia por valor realizada por cloneEvent garante que futuras retransmissões conterão a mensagem recebida da parte superior da pilha por esta camada. Se, pelo contrário, fosse realizada uma cópia por referência, à mensagem mantida pelo temporizador seriam adicionados os cabeçalhos das camadas inferiores, impossibilitando retransmissões posteriores. A linha 9 altera a referência para a sessão que criou o evento e o novo caminho do evento é calculado na linha 10. O evento contendo a mensagem é retransmitido através da invocação do método go da linha 11.

Da linha 12 à linha 14 é apresentada uma possível aplicação das facilidades de depuração de erros do *Appia*. O atributo DEBUGOUTPUT da classe FIFOSESSION é

```
/**
 * Main Event handler function. Accepts all incoming events and
 * dispatches them to the appropriate functions
 * @ params e The incoming event
 * @ see Session
 */

public void handle(Event e) {
    if (e instanceof AckEvent)
        handleAck((AckEvent)e);
    else if (e instanceof ChannelInit)
        handleChannelInit((ChannelInit)e);
    else if (e instanceof FIFOTimer)
        handleFIFOTimer((FIFOTimer)e);
    else if (e instanceof SendAckTimer)
        handleSendAckTimer((SendAckTimer)e);
    else if (e instanceof SendableEvent)
        handleSendable((SendableEvent)e);
    else if (e instanceof Debug)
        handleDebug((Debug)e);
    else if (e instanceof MaxPDUSizeEvent)
        handlePDUSize((MaxPDUSizeEvent)e);
    else if (e instanceof FIFOConfigEvent)
        handleConfigEvent((FIFOConfigEvent)e);
    else {
        /* Unexpected event arrived */
        try {
            e.go();
        }
        catch(AppiaEventException ex) {}
    }
}
```

Figura 5.2: Excerto da concretização da classe FifoSession

inicializado a NULL e poderá conter uma referência para a OUTPUTSTREAM recebida como atributo de um evento DEBUG. Sempre que DEBUGOUTPUT apresentar um valor diferente de NULL, o protocolo utiliza-o como destino das mensagens descrevendo as situações consideradas relevantes. Uma das optimizações do compilador Java é não introduzir no *bytecode* instruções que não serão executadas. O atributo DEBUGON é uma constante booleana que explora essa característica para que a informação de depuração não prejudique o desempenho dos protocolos concluídos. Uma vez que o valor `falso` é o elemento absorvente da disjunção, a declaração da constante com esse valor equivale a omitir, do código compilado, todas as condições semelhantes à apresentada.

Entre as linhas 17 e 24 é solicitada uma nova temporização, que será associada à mensagem entretanto retransmitida. A utilização da mesma instância do objecto FI-

```
private void resend(FIFOTimer timeout) {
    /* Resend message. FIFOTimeout what's attribute contains an WaitingMessage
       object which in turn contains a copy of the unacknowledged message. Make another
       copy and tell it to go again. */
    try {
        SendableEvent unack = (SendableEvent)timeout.getWait().e.cloneEvent();

        /* The copy is not a "real" clone. Generator has changed */
        unack.setSource(this);
        unack.init ();
        unack.go();
        if (debugOn && debugOutput!=null)
            debugOutput.println ("FIFO: Unacknowledged message with sequence number "+
                                 timeout.getWait().seqNumber +" resent");

        /* Also perform a duplication of the timeout to request a new one */
        FIFOTimer newtimeout=timeout;
        newtimeout.retry++;
        newtimeout.setTimeout(System.currentTimeMillis()+fifotimeout);
        newtimeout.setDirection(new Direction(Direction.DOWN));
        newtimeout.setSource(this);
        newtimeout.getQualifier().set(EventQualifier.ON);
        newtimeout.init();
        newtimeout.go();
        if (debugOn && debugOutput!=null)
            debugOutput.println ("FIFO: New timeout set");
    }
    catch(CloneNotSupportedException ex) {
        System.err.println ("FIFO: Clone not supported exception");
    }
    catch(AppiaEventException ex) {
        switch(ex.type) {
        case AppiaEventException.UNWANTEDEVENT :
            System.err.println ("Unwanted event exception catched in FifoSession");
            break;
        case AppiaEventException.UNKNOWNSESSION :
            System.err.println ("Unknown session exception catched in FifoSession");
            break;
        case AppiaEventException.ATTRIBUTEMISSING :
            System.err.println ("Missing attribute exception catched in FifoSession");
            break;
        case AppiaEventException.NOTINITIALIZED :
            System.err.println ("Impossible exception event not initialized in FifoSession");
            break;
        }
    }
    catch(AppiaException ex) {
        System.err.println ("Negative time in event definition");
    }
}
```

Figura 5.3: Método que procede ao reenvio de mensagens não confirmadas

FOTIMER permite economizar recursos de libertação e reserva de memória. No entanto, o evento enviado para o canal (linha 24) é, para o *Appia*, um novo evento uma vez que teve todos os atributos redefinidos e foi inicializado. O evento será enviado no sentido descendente do canal com o qualificador ON, solicitando por isso ao canal que active um novo temporizador (o evento foi recebido com o qualificador NOTIFY).

A partir da linha 28 são tratadas as excepções levantadas pelos métodos invocados. Dados os custos elevados que representam para o sistema, as excepções não foram utilizadas como ferramenta de controlo de fluxo.

O método apresentado é um exemplo das potencialidades de utilização dos protocolos num número indefinido de canais, característica do *Appia*. O identificador de canal é sempre utilizado como um valor opaco. O canal por onde o evento foi inicialmente recebido é utilizado, neste protocolo, para o pedido dos temporizadores e para a retransmissão do evento, caso se mostre necessário.

O protocolo FIFO pode ser configurado e colabora na configuração, em tempo de execução, de outros protocolos. A sua informação de configuração é recebida através de eventos da classe FIFOCONFIGEVENT, representada na figura 5.4. Os parâmetros configuráveis são o tempo que precede o envio de uma confirmação, a dimensão da janela, o número de retransmissões e o intervalo entre elas. De notar que estão definidos valores por omissão para estas opções e que a sua alteração pode ocorrer por mais de uma vez durante a execução. Os valores definidos são independentes entre sessões o que aumenta a capacidade de adaptação do protocolo a canais com diferentes requisitos.

5.1.2 Composição de protocolos

Como foi já referido, um canal é construído pela instanciação de uma Qualidade de Serviço. Esta, por sua vez, é definida por um vector de camadas. A construção mais simples de um canal usa as instruções apresentadas na figura 5.5.

As quatro primeiras linhas definem um vector contendo referências para as camadas que participarão na qualidade de serviço. Na linha 6 é definida uma nova quali-

```
public class FIFOConfigEvent extends Event {
    public FIFOConfigEvent(Channel c, Direction d, Session source)
                                        throws AppiaEventException;
    public void setRetries(int nRetries);
    public boolean isRetriesDef();
    public int getRetries ();
    public void setPeriod(int period);
    public boolean isPeriodDef();
    public int getPeriod();
    public void setWindow(int window);
    public boolean isWindowDef();
    public int getWindow();
    public void setPiggyback(int piggyback);
    public boolean isPiggybackDef();
    public int getPiggyback();
}
```

Figura 5.4: Assinatura do evento FIFOConfigEvent

```
Layer[] layersSimples=new Layer[3];
layersSimples[0]=new UDPSimpleLayer();
layersSimples[1]=new FIFOLayer();
layersSimples[2]=new ApplicationLayer();
try {
  QoS qossimples=new QoS ("QoSSimples",layersSimples);
}
catch(AppiaInvalidQoSException ex) {
  System.err.println ("Combinacao invalida de protocolos");
  System.exit(1);
}
Channel chanSimples=qossimples createUnboundChannel ("CanalSimples");
chanSimples.start();
```

Figura 5.5: Construção elementar de um canal dade de serviço. O construtor irá proceder à validação da composição, assegurando-se de que todos os eventos necessários são criados por alguma camada. A não satisfação desta condição conduzirá o fluxo do programa para o tratamento da excepção da linha 8. Em seguida, o construtor define os caminhos de eventos da QoS.

A linha 12 solicita à qualidade de serviço que defina um canal que a satisfaça. A linha 13 inicia efectivamente o canal. Uma vez que o código não define qualquer sessão para o canal, todas as sessões que participarão no canal serão atribuídas de forma automática. Após a atribuição, o canal cria caminhos de eventos onde as camadas da QoS são substituídas por referências para as sessões.

5.1.2.1 Grupos ligeiros

O modelo de sincronia virtual para a comunicação em grupo provou ser um paradigma importante na construção de aplicações distribuídas. Concretizações deste modelo tipicamente requerem a utilização de detectores de faltas e de protocolos de recuperação. Em aplicações que requerem a utilização de um número significativo de grupos, são conseguidos ganhos significativos de desempenho se estes grupos partilharem os recursos necessários à obtenção de sincronia virtual. Um serviço que projecta grupos do utilizador em instâncias de uma concretização de sincronia virtual é denominado um serviço de grupos ligeiros (Rodrigues *et al.*, 1996).

Os grupos ligeiros podem ser concretizados de duas formas, com diferentes impactos no desempenho. No modelo estático, a projecção entre um grupo de utilizador e um grupo de baixo nível é fixa. Pelo contrário, quando ocorre comutação dinâmica, o serviço altera as projecções estabelecidas, adaptando-as às mudanças que ocorrem na filiação de ambas as classes de grupos. O consumo de recursos pode ser substancialmente diminuido quando é utilizada comutação dinâmica. O *Appia* suporta ambos os modelos. Para manter a clareza da exposição, o ênfase é colocado na concretização estática do serviço.

Um esboço do código que concretiza a versão estática do serviço de grupos ligeiros é apresentado nas figuras 5.6 e 5.7 e serve também como exemplo de atribuição explícita e automática.

```
1  Channel novoCanal=aQoS.createUnboundChannel("CanalUsandoLWG");
2  ChannelCursor cc=novoCanal.getCursor();
3  cc.top();
4  try {
5    while(cc.getLayer()!=lwgLayer) {
6      cc.down();
7    }
8    cc.setSession(lwgSession);
9  }
10 catch(AppiaCursorException ex) {
11   System.err.println("Operacao incorrecta ao cursor do canal");
12   System.exit(1);
13 }
14 novoCanal.start();
```

Figura 5.6: Esboço do código para definição de um canal utilizando grupos ligeiros

A atribuição explícita está representada na figura 5.6: após a criação do canal, que ocorre na linha 1, a aplicação solicita um cursor (linha 2) que utiliza para localizar a posição que deve ser preenchida por uma sessão criada pela camada referida na variável lwgLayer (linhas 4 a 7). Essa posição vê-lhe explicitamente atribuída a sessão referida pela variável lwgSession, na linha 8. A responsabilidade do processo de atribuição de sessões passa para a responsabilidade do canal na linha 14, com a invocação do método start. Os valores de lwgLayer e lwgSession podem ter sido obtidos de duas formas: questionando um canal já definido através dos métodos getLayer e getSession da classe CHANNELCURSOR ou mantendo as referências em variáveis.

A figura 5.7 apresenta o método boundSessions da sessão de grupos ligeiros para exemplificar um algoritmo de atribuição automática de sessões. Depois de verificar que já existe uma projecção para o grupo que o canal irá subscrever, o método copia todas as sessões do canal modelo que se encontrem abaixo de si na pilha. Realiza para isso dois ciclos. O primeiro, da linha 11 à linha 14 procura a referência para si próprio em ambos os canais. As sessões acima desta camada não pertencem ao grupo de baixo nível e são ignoradas. O segundo ciclo, que se inicia na linha 15 e termina na linha 21 copia todas as sessões do canal já existente para o canal em definição. Na prática, esta operação define para o novo canal o mesmo grupo de baixo nível que estava a ser utilizado no canal modelo.

A concretização dinâmica do serviço implica a alteração da composição de um

```
public void boundSessions(Channel newChannel) {
  Channel modelo=procuraGrupo(newChannel.getChannelID());
  if (modelo!=null) {
    /* Ja existe uma projeccao do grupo ligeiro num qualquer grupo de baixo nivel.
       Copia todas as sessoes para o novo canal */
    ChannelCursor ccModelo=modelo.getChannelCursor(),
                  ccNovo=newChannel.getChannelCursor();
    ccModelo.top();
    ccNovo.top();
    try {
      while(ccModelo.getSession()!=this) {
        ccModelo.down();
        ccNovo.down();
      }
      while(ccNovo.isPositioned()) {
        if (ccNovo.getSession()==null) {
          ccNovo.setSession(ccModelo.getSession());
        }
        ccNovo.down();
        ccModelo.down();
      }
    catch(AppiaCursorException ex) {
      System.err.println ("Acesso invalido ao cursor do canal");
      System.exit (1);
    }
  }
  /* O grupo ligeiro ainda nao existe . Esta sessao projecta o novo
     grupo num canal adequado. O codigo relevante seria equivalente. */
}
```

Figura 5.7: Código de atribuição automática de camadas da sessão do protocolo de grupos ligeiros.

```
public static void main(String args[]) {
  Layer intermediaLayer=new IntermediaLayer();
  Session intermediaSession=intermediaLayer.createSession();
  intermediaSession.makeStack();
  Appia.run();
}
```

Figura 5.8: Inicialização da aplicação de sincronização de dados multimédia

canal durante a sua execução, assegurando que as sessões e os eventos têm uma transição consistente. Esta garantia pode ser obtida bloqueando temporariamente o canal através da invocação do seu método end. Após a alteração da composição, o método start calcula os novos caminhos de eventos e sinaliza às sessões o reinício da actividade.

5.1.2.2 Sincronização de dados multimédia

A secção 4.1.1, na página 54, apresenta um exemplo de composição para uma aplicação multimédia que integra três canais de dados distintos, para troca de texto, audio e vídeo. A solução apresentada delega na camada especializada INTERMEDIASYNC a construção da composição apresentada na figura 4.2.

As instruções apresentadas na figura 5.8 começam por criar um objecto da classe IntermediaLayer ao qual é solicitada a criação de uma sessão. A esta será depois invocado o método makeStack, apresentado na figura 5.9, que compõe e inicia os canais de audio, vídeo e texto. Uma vez que os canais estão definidos e registados no núcleo do *Appia*, o executável pode invocar o método run, entregando-lhe o controlo do processo.

O método *makeStack* (figura 5.9) realiza quatro operações distintas: definição das qualidades de serviço (linhas 2 a 18), definição dos canais (linhas 20 a 23), atribuição explícita de sessões (linhas 25 a 47) e inicialização dos canais (linha 48).

São definidas três qualidades de serviço distintas, uma para cada um dos tipos de dados utilizados pela aplicação. O vector contendo a descrição da qualidade de serviço de vídeo, definido entre as linhas 3 e 6, cria instâncias das camadas comuns às três qualidades de serviço. Os dois vectores restantes utilizam essas referências para impedir que existam diversas instâncias de uma mesma classe LAYER. Isso poderia

```
public void makeStack() {
  /* Definicao de QoSs */
  Layer[] videoArray={
            new UdpSimpleLayer(),new FailureDetectLayer(),new VideoLayer(),
            new FifoLayer(),getLayer(),new ApplicationLayer()
      };
  Layer[] audioArray={
            videoArray[0],videoArray[1],new AudioLayer(),videoArray[3],videoArray[4],
            videoArray[5]
      },
          dataArray={
            videoArray[0],videoArray[1],new DataLayer(),videoArray[3],videoArray[4],
            videoArray[5]
      };

  QoS videoQoS = new QoS ("QoS video",videoArray),
       audioQoS = new QoS ("QoS audio",audioArray),
       dataQoS = new QoS ("QoS dados",dataArray);

  /* Definicao dos canais. */
  videoChannel = videoQoS.createUnboundChannel ("canal video");
  audioChannel = audioQoS.createUnboundChannel ("canal audio");
  dataChannel = dataQoS.createUnboundChannel ("canal de dados");

  /* Atribuicoes explicitas */
  ChannelCursor videocc = videoChannel.getChannelCursor(),
                audiocc = audioChannel.getChannelCursor(),
                datacc  = dataChannel.getChannelCursor();
  videocc.bottom(); audiocc.bottom(); datacc.bottom();
  try {
    Session udpSession=videoArray[0].createSession();
    videocc.setSession(udpSession); audiocc.setSession(udpSession); datacc.setSession(udpSession);
    videocc.up (); audiocc.up (); datacc.up ();
    Session fdSession=videoArray[1].createSession();
    videocc.setSession(fdSession ); audiocc.setSession(fdSession ); datacc.setSession(fdSession);
    videocc.up (); audiocc.up (); datacc.up ();
    videocc.up (); audiocc.up (); datacc.up ();
    videocc.up (); audiocc.up (); datacc.up ();
    videocc.setSession( this ); audiocc.setSession( this ); datacc.setSession( this );
    videocc.up (); audiocc.up (); datacc.up ();
    Session applSession=videoArray[5].createSession();
    videocc.setSession(applSession); audiocc.setSession(applSession); datacc.setSession(applSession);
  }
  catch(AppiaCursorException ex) {
    System.err.println ("Operacao invalida aos cursores");
    System.exit (1);
  }
  videoChannel.start (); audioChannel.start (); dataChannel.start ();
}
```

Figura 5.9: Excerto do código que concretiza uma composição de sincronização de dados multimédia afectar, por exemplo, operações de comparação de qualidades de serviço, baseadas na comparação das referências para os objectos. As qualidades de serviço estão definidas entre as linhas 16 e 18.

A linha 25 inicia a atribuição explícita de sessões aos canais. Uma única sessão dos protocolos UdpSimpleSession, FailureDetect, IntermediaSync e Application é partilhada por todos os canais. A atribuição destas camadas é apresentada entre as linhas 31 e 42 da figura. O processo é idêntico para todas as atribuições e começa pela criação de uma sessão, solicitada ao objecto da classe LAYER respectiva. A referência obtida é depois utilizada como argumento aos métodos setSession de cada um dos cursores de canal. As camadas não partilhadas são omitidas. Aquando da execução do método start (linha 48) em cada um dos canais, e se não houver sessões que realizem atribuição automática, o canal realizará a atribuição por omissão que resultará na criação de novas instâncias das camadas.

5.2 Desempenho

O desempenho do *Appia* foi avaliado em duas vertentes. A primeira analisa o impacto das opções tomadas na fase de desenho e concretização, a segunda o desempenho da plataforma com uma composição simples de protocolos.

As medições foram realizadas utilizando dois computadores equipados com processadores Pentium III a 500MHz. O sistema operativo utilizado foi o Windows NT. O *Appia* foi executado sobre uma máquina virtual Java versão 1.2.2. As estações de trabalho estão ligadas sobre uma rede local Ethernet a 10Mb/s com pouca utilização.

5.2.1 Mensagens

Na concretização da classe MESSAGE, o esforço de optimização foi centrado nas operações mais usuais: adição e remoção de cabeçalhos e cópia da mensagem para um vector contíguo de octetos.

Uma primeira aproximação ao problema passou por concretizar em Java o interface da biblioteca de mensagens utilizada no x-Kernel (Mosberger, 1996). Os resultados obtidos com esta versão inicial foram insatisfatórios, sobretudo pela impossibilidade do Java em aceder directamente a endereços de memória. Houve então necessidade de adaptar o modelo inicial às características da linguagem, sacrificando alguma da simplicidade de utilização. A tabela 5.1 apresenta os resultados da avaliação de desempenho da biblioteca de mensagens.

Operação	Resultado (μs)		
	10 octetos	100 octetos	1000 octetos
push	2,864	4,278	14,683
pop	2,599	2,605	2,802

Tabela 5.1: Desempenho das operações de adição e remoção de cabeçalhos

Os testes à operação push criam mensagens com 10000 octetos realizando operações que adicionam, respectivamente 10, 100 e 1000 octetos de cada vez. Cada bloco reserva 1000 octetos pelo que, em qualquer um dos testes, são criados 10 blocos. O tempo médio de adição de um cabeçalho de 1000 octetos é cerca de 5 vezes superior ao de um cabeçalho com 10 octetos. Esta desproporção é justificada pelo número constante de operações de reserva de blocos. Nos cabeçalhos de menor dimensão, as 100 operações necessárias para o preenchimento de um bloco atenuam o impacto que esta operação apresenta no desempenho final. Pelo contrário, cada operação push de 1000 octetos adiciona um bloco à mensagem. A reserva de blocos mostra-se por isso o ponto de estrangulamento da operação.

Os testes à operação pop esvaziam uma mensagem de 10000 octetos em operações que retiram, respectivamente 10, 100 e 1000 octetos. A mensagem encontra-se num único bloco, definido com a dimensão total da mensagem. Este será o comportamento usual em tempo de execução: a sequência de octetos recebida da rede é utilizada na definição de um único bloco, que constituirá a mensagem.

As chamadas que enviam datagramas para a rede exigem que os dados a transmitir se encontrem num vector contíguo de octetos. Ao atingirem a última camada da composição, as mensagens têm por isso que converter a sequência de blocos que entretanto foram criando num único vector. A dimensão das mensagens desempenha aqui um papel fundamental por condicionar o número de blocos definidos e, consequentemente, a quantidade de octetos a copiar. A biblioteca dispende $161\mu s$ a copiar uma mensagem de 10000 octetos, composta por 10 blocos com 1000 octetos cada.

5.2.2 Impacto dos caminhos de eventos

Uma das características inovadoras do *Appia* é o facto de cada sessão processar apenas os eventos que subscreveu. Para medir o impacto dos caminhos de eventos no desempenho do sistema, foram definidos os protocolos TRANSPARENTE e FANTASMA. O primeiro não aceita qualquer evento. O segundo aceita todos os eventos mas não realiza qualquer operação sobre eles. Os eventos recebidos pelo protocolo FANTASMA são imediatamente encaminhados para a sessão seguinte.

Instâncias destes protocolos foram combinadas em canais com o protocolo TESTE, o qual introduz no canal sequências de eventos do tipo ECHOEVENT. Como referido na secção 4.3.2.1, quando um ECHOEVENT atinge o fim do canal, o evento que transporta é introduzido no canal no sentido oposto. As sessões de TESTE medem o tempo entre a introdução do ECHOEVENT e a recepção do evento transportado.

	Tempo de retorno dos eventos (μs)				
n^o de sessões	0	1	2	3	10
Transparente	19,61	19,63	19,51	19,64	19,54
Fantasma	19,61	29,75	39,95	52,13	140,38

Tabela 5.2: Atraso sofrido pelos eventos em canais com protocolos TRANSPARENTE e FANTASMA

A tabela 5.2 mostra que, como se esperava, a adição do protocolo TRANSPARENTE não afecta o desempenho do canal. As oscilações apresentadas ao longo da primeira linha da tabela são inferiores a 0,04% do tempo médio. Por essa razão, supõe-se serem resultantes de eventos externos ao *Appia* e são desprezadas.

Por sua vez, cada sessão do protocolo FANTASMA introduz, em média $10,80\mu s$ ao processamento de um evento. Estes valores justificam a utilidade dos caminhos de eventos do *Appia*.

5.2.3 Desempenho global

Para avaliar a taxa de transferência do *Appia* foi definida uma pilha com três protocolos que, compostos, providenciam um serviço básico de transmissão fiável de datagramas. Os protocolos utilizados foram:

UDPSimple : acesso à rede utilizando o protocolo UDP;

FIFO : ordenação e entrega fiável de mensagens;

Appl : interface em linha de texto com o utilizador;

Para avaliação do desempenho, uma das camadas Appl envia uma mensagem de 90 octetos para o seu par remoto, o qual a devolve imediatamente. Sequências de 20 destas operações foram realizadas para estimar um valor médio. O tempo de ida e volta de uma mensagem foi em média de $2,5ms$.

Para estabelecer uma base de comparação, foi desenvolvida, também na linguagem Java, uma outra aplicação. Esta realiza a mesma operação utilizando apenas um ponto de acesso à rede disponibilizado pela máquina virtual Java. O tempo de ida e volta médio foi, utilizando o protocolo UDP, de $0,44ms$ e de $0,574ms$ utilizando o protocolo TCP. Destes resultados é possível tirar duas conclusões.

A utilização do *Appia* adiciona $2,06ms$ ao tempo de ida e volta de uma mensagem. Este valor resulta da subtracção do tempo obtido pelo protocolo UDP ao valor medido utilizando o *Appia*. Uma vez que cada mensagem percorre a composição quatro vezes, cada uma contribui com cerca de $0,515ms$ para o tempo de ida e volta de uma mensagem.

Competindo com o protocolo TCP, que fornece propriedades semelhantes, o *Appia* apresenta um desempenho inferior em cerca de $1,926ms$ por ida e volta de uma mensagem.

Não é possível encontrar na bibliografia uma comparação suficientemente sólida que permita avaliar, em termos absolutos, o desempenho do *Appia*.

Uma mensagem demora, no x-Kernel, $2,00ms$ a percorrer num sentido uma composição com o protocolo UDP e $3,30ms$ com o protocolo TCP (Hutchinson & Peterson, 1991). No entanto, a diferença de poder computacional torna inviável qualquer comparação: o x-Kernel apresentava um desempenho superior às concretizações dos mesmos protocolos no sistema operativo Unix.

O Horus apresenta resultados onde o envio de uma mensagem sem ordenação e utilizando o protocolo de sincronia virtual demora $1,2ms$ (van Renesse *et al.*, 1996). Estes resultados são próximos dos obtidos no *Appia*: extrapolando o valor apresentado, pode-se concluir que o tempo de ida e volta seria aproximadamente $2,4ms$. No entanto, de acordo com a Lei de Moore (Moore, 1965) o poder computacional terá já quadruplicado pelo que o desempenho do Horus seria actualmente bastante superior.

Das plataformas analisadas, a mais recente foi o Ensemble. Nesta, o tempo dispendido por uma mensagem a percorrer uma composição que oferece sincronia virtual com ordenação FIFO foi de $1,5ms$ (Hayden, 1998). Este valor foi obtido para a versão não optimizada da plataforma. Aplicando os trilhos de eventos e o adiamento de operações, os valores baixam para $41\mu s$ na linguagem ML e $26\mu s$ no caso da linguagem C. Esta é a comparação que mais penaliza o *Appia*, uma vez que o núcleo do Ensemble apresenta um desempenho mais de dez vezes superior para um conjunto mais rico de propriedades.

5.2.3.1 Factores que penalizam o desempenho

O primeiro objectivo do *Appia* foi oferecer um ambiente inovador para a composição de protocolos. As optimizações ao desempenho encontram-se ainda numa fase embrionária. Os testes realizados utilizaram versões iniciais dos protocolos, concretizadas para teste e demonstração das propriedades do núcleo. O trabalho futuro passará certamente pelo estudo de optimizações aos algoritmos utilizados na concretização dos protocolos.

Uma outra condicionante do desempenho é a linguagem de concretização. Para além dos problemas característicos numa linguagem interpretada, não se pretendeu nesta fase realizar um trabalho exaustivo no sentido de garantir a utilização das ferramentas que melhor exploram as características da linguagem. A principal prioridade foi a de conseguir um ambiente correcto, onde fosse possível ao programador assimilar rapidamente a interface para a concretização de protocolos e aplicações.

5.3 Aplicações do sistema

O *Appia* tem registado uma boa receptividade junto da comunidade científica. Para além das publicações que regista, o sistema está neste momento integrado em quatro projectos, qualquer um deles envolvendo investigadores não pertencentes ao Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa.

MOOs (Multi-user Object-Oriented environments) são ambientes orientados aos objectos onde múltiplos utilizadores interagem síncrona e assincronamente no contexto de mundos virtuais. O objectivo do projecto Moosco (Multi-User Object-Oriented environments with Separation of COncerns) é aplicar a separação de conceitos às abstracções de domínio dos MOOs e à arquitectura distribuída. No contexto deste projecto, o *Appia* está a ser utilizado como sistema de suporte à distribuição. Simultaneamente, o modelo de composição está também a ser analisado. Pretende-se avaliar até que ponto a composição protocolar pode ser extendida para outros domínios, neste caso, a gestão da replicação. O projecto Moosco é realizado conjuntamente com o Grupo de Engenharia de Software do Instituto de Engenharia de Sistemas e Computadores (INESC) e financiado pela Fundação da Ciência e Tecnologia, no âmbito do programa Praxis XXI. O investigador responsável é o Professor António Rito da Silva, do INESC.

O projecto SHIFT (Semantic HInts on Fault Tolerance) estuda a aplicação de informação semântica às mensagens para determinar a sua obsolescência. Num ambiente de comunicação em grupo onde as mensagens são frequentemente inutilizadas pela chegada de outras, a eliminação das mensagens mais antigas e ainda não apresentadas à aplicação em alguns dos membros permite atenuar as diferenças de velocidade de processamento. Este sistema tem aplicações, por exemplo, na leitura de sensores e na difusão de cotações da bolsa de valores. Uma das tarefas do projecto consiste em analisar os requisitos que os sistemas de suporte à distribuição terão que satisfazer. Actualmente, estuda-se a possibilidade de adaptar o *Appia* ao modelo. O projecto SHIFT é realizado conjuntamente com o Grupo de Sistemas Distribuídos da Universidade do Minho e financiado pela Fundação da Ciência e Tecnologia, no âmbito do programa Praxis XXI. O investigador responsável é o Professor Luís Rodrigues, da Universidade de Lisboa.

Dois projectos internacionais avaliam presentemente a utilização do *Appia* como sistema de suporte à distribuição. O projecto MAFTIA investiga o paradigma da tolerância a faltas para propor uma arquitectura de sistema integrada, construída sobre este paradigma. O *Appia* está presentemente a ser considerado como plataforma de concretização dos protocolos. O projecto é coordenado pela Universidade de Newcastle upon Tyne e são parceiros: a Faculdade de Ciências da Universidade de Lisboa, a Defence Evaluation and Research Agency (DERA), a Universidade de Saarlandes, o Laboratory for Analysis and Architecture of Systems (LAAS) e a IBM Zurique. O financiamento é proveniente da Comissão Europeia, ao abrigo do programa IST.

O *Appia* integra-se ainda no projecto Globdata cujo objectivo é conceber e produzir uma ferramenta de desenvolvimento aplicacional eficiente e um sistema de suporte denominados COPLA. O COPLA oferecerá aos programadores uma visão global de um repositório persistente de objectos. O repositório estará geograficamente distribuído e terá acesso transaccional. O Globdata é um projecto financiado pelo programa IST da Comissão Europeia, coordenado pelo Instituto Tecnológico de Informática de Valência (ITI). Os parceiros de investigação no consórcio são: o Instituto Tecnológico de Informática de Valência (ITI), a Faculdade de Ciências da Universidade de Lisboa e a Universidade Pública de Navarra (UPNA). Os parceiros industriais, responsáveis pela possível comercialização do produto final são: a GFI Informatique e a Investigación y Desarrollo Informático (IDI EIKON).

5.4 Sumário

Este capítulo avalia a concretização do *Appia*. São abordadas duas vertentes: a capacidade de resolver os problemas anteriormente enunciados e para os quais as plataformas existentes não apresentam soluções suficientemente genéricas e o desempenho do sistema.

Para mostrar a independência entre a concretização dos protocolos e as composições em que participam são apresentados exemplos do código desenvolvido. Estes permitem também avaliar a simplicidade da interface do *Appia*. O sucesso na concretização de composições complexas sem prejudicar a reusabilidade dos protocolos é exemplificado, por esboços de soluções para os casos apresentados na secção 4.1. Para ambos são propostas composições que satisfazendo os requisitos não comprometem a independência de alguns dos protocolos participantes.

As medições de desempenho avaliaram três factores: a classe de suporte a mensagens, as vantagens do modelo de encaminhamento de eventos e o desempenho global do sistema. A criação de caminhos de eventos mostrou-se uma vantagem do *Appia*. O seu impacto no desempenho final do sistema é suficientemente importante para que seja registado como uma característica que só pode beneficiar qualquer ambiente de suporte à composição de protocolos.

Contudo, o *Appia* não apresenta um desempenho notável quando confrontado com as medições apresentadas por outros sistemas. A utilização da linguagem Java e as linhas orientadoras da codificação são apontadas como as principais responsáveis pelos valores obtidos.

Notas

As primeiras aproximações da aplicação do *Appia* no suporte a ambientes virtuais multi-utilizador, experimentada no projecto Moosco resultaram na publicação, até à conclusão desta dissertação, de dois artigos. Nas actas do "Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)", realizado em conjunto com o "19th IEEE Symposium on Reliable Distributed Systems" (Miranda *et al.*, 2000a) e nas actas do "International Workshop on Distributed Dynamic Multiservice Architectures", realizado em simultâneo com a 21ª Conferencia Internacional de Sistemas Computacionais Distribuídos (ICDCS'21) a realizar em Abril de 2001 (Antunes *et al.*, 2001).

Os resultados preliminares de uma possível extensão do *Appia* para plataformas de tempo-real foram publicados em (Rodrigues *et al.*, 2001).

O código fonte e a documentação do *Appia* estão disponíveis para utilização não comercial no endereço `http://appia.di.fc.ul.pt`. O sistema tem sido utilizado no suporte a projectos das cadeiras de Protocolos de Redes de Dados e Tolerância a Faltas Distribuída da Licenciatura em Informática da Faculdade de Ciências da Universidade de Lisboa.

6 Conclusões e trabalho futuro

A decomposição dos protocolos de comunicação de redes de computadores em camadas, cada uma proporcionando um conjunto reduzido de propriedades, apresenta vantagens e desvantagens. As vantagens deste modelo de programação são: a redução dos tempos de desenvolvimento e depuração e a simplificação das operações de manutenção e actualização. No entanto, a estruturação que lhe é imposta conduz a um código menos eficiente. O aumento do número de chamadas a procedimentos e a redundância de operações são algumas das desvantagens.

O conjunto de ambientes de suporte à composição de protocolos é já bastante completo. Cada um oferece uma combinação diferente de entre um conjunto de propriedades como o momento de composição, o encaminhamento das mensagens ou a política de escalonamento. Com esta dissertação pretendeu-se analisar e confrontar as opções de alguns dos mais relevantes.

Com raras excepções, todos os sistemas assumem que os protocolos são compostos verticalmente. Nestes, encontram-se duas variações: aqueles que permitem a diferentes composições partilhar sessões e aqueles onde nenhuma sessão é partilhada. Todos os que assumem o primeiro modelo impõem contudo uma restrição adicional: a impossibilidade, de forma transparente para os protocolos, de criar combinações (canais) onde existam sessões partilhadas intermediadas por sessões não partilhadas. A dissertação apresenta dois exemplos distintos, através dos quais se demonstra a utilidade destas "composições irregulares".

Idealmente, os protocolos devem ser capazes de exprimir diversos acontecimentos. Uma das formas adoptadas é o recurso a eventos: estruturas de dados caracterizadas por um tipo. A abordagem dos sistemas aos eventos é variável. Alguns definem um conjunto limitado de tipos de eventos, que todos os protocolos terão que saber interpretar; outros entregam aos programadores a responsabilidade de definir quais os tipos adequados. Nesse caso, os tipos utilizados são transparentes para o sistema. Nenhuma das soluções é perfeita, sendo as vantagens de um os pontos fracos do outro. Se o sistema definir os eventos, poderá acompanhar a comunicação e intervir, melhorando o desempenho e simplificando o código a executar pelos protocolos. Por outro lado, torna-se mais difícil extender o sistema a configurações não antecipadas pelos autores, onde poderá ser necessário um conjunto de eventos mais rico do que o inicialmente previsto.

A dissertação apresenta um sistema de suporte à composição denominado *Appia*, que apresenta algumas contribuições inovadoras:

- Propõe um modelo alternativo de encaminhamento, onde os eventos visitam apenas as camadas que efectuarão operações úteis em resultado da sua recepção.
- Alarga os modelos de composição, ao suportar composições irregulares, onde as sessões podem ser partilhadas independentemente da sua posição.
- Utiliza um sistema misto de eventos, onde todos os tipos de eventos, inclusive os definidos pelo núcleo, podem ser extendidos para conter informação mais completa. Esta aproximação não compromete a reutilização dos protocolos em outras combinações, onde coopera com protocolos de versões posteriores àquelas para as quais foi concebido.

O trabalho futuro será centrado em duas vertentes: melhoramentos ao núcleo do *Appia* e desenvolvimento e aperfeiçoamento de protocolos.

Durante a fase de desenho, houve a preocupação em evitar opções que limitassem as possibilidades de experimentação ou configuração do sistema. O *Appia* deixou por isso em aberto um conjunto de possibilidades que não foram ainda experimentadas e que poderão trazer vantagens ao sistema.

Um exemplo é o modelo de concorrência a utilizar. Sem alterações ao núcleo, cada escalonador de eventos pode ter a sua própria actividade ou partilhá-la com outros escalonadores. Com pequenas alterações, é possível concretizar o modelo de um processo ligeiro por evento.

Os protocolos foram codificados tendo em vista dois objectivos: confrontar a interface oferecida pelo *Appia* com os requisitos reais destas entidades e servir como demonstradores a observadores externos (idealmente, futuros utilizadores do sistema). Para além das facilidades oferecidas pela plataforma, não houve uma preocupação excessiva em garantir que a versão utilizada seria aquela que apresentaria o melhor desempenho. Mantendo-se as perspectivas de utilização do *Appia*, este será necessariamente um ponto a desenvolver. Passada a fase inicial de validação, os utilizadores exigirão garantias de eficiência do sistema e dos protocolos que o compõem.

O grande desafio do *Appia* ocorrerá no futuro próximo com a sua inclusão em dois projectos internacionais: na área da segurança de sistemas informáticos (projecto MAFTIA), que visa a construção de um ambiente tolerante a intrusões e na das bases de dados distribuídas de grande escala, com o projecto Globdata. A definição de protocolos para ambos bem como os requisitos que vierem a ser solicitados, potenciarão o amadurecimento do projecto. O *Appia* pode assim vir a tornar-se um ambiente de desenvolvimento e investigação com projecção significativa.

Bibliografia

ANTUNES, MIGUEL, MIRANDA, HUGO, SILVA, ANTÓNIO RITO, RODRIGUES, LUÍS, & MARTINS, JORGE. 2001. Separating Replication from Distributed Communication: Problems and Solutions. *Page to appear of: Proceedings of the International Workshop on Distributed Dynamic Multiservice Architectures*. Phoenix, Arizona, USA: IEEE.

BHATTI, NINA T., HILTUNEN, MATTI A., SCHLICHTING, RICHARD D., & CHIU, WANDA. 1998. Coyote: A System for Constructing Fine-grain Configurable Communication Services. *ACM Transactions on Computer Systems*, **16**(4), 321–366.

BIRMAN, K., & VAN RENESSE, R. (eds). 1994. *Reliable Distributed Computing With the ISIS Toolkit*. IEEE CS Press.

BIRMAN, K., FRIEDMAN, R., & HAYDEN, M. 1997 (Feb.). *The Maestro Group Manager: A structuring Tool For Applications With Multiple Quality of Service Requirements*. Tech. rept. Cornell University, Ithaca, USA.

BIRMAN, KENNETH P. 1996. *Building secure and reliable network applications*. Manning Publications Co.

CLARK, D. D. 1982 (July). *Modularity and efficiency in protocol implementation*. Request for Comments 817. MIT Laboratory for Computer Science, Computer Systems and Communication Group.

CLARK, D. D., & TENNENHOUSE, D. L. 1990 (Sept.). Architectural considerations for a new generation of protocols. *Pages 200–208 of: Proceedings of the ACM symposium on Communications architectures & protocols*.

CORREIA, M., & PINTO, P. 1995. Low-level multimedia synchronization algorithms on broadband networks. *Pages 423–434 of: The Third ACM Intl. Multimedia Conference and Exhibition (MULTIMEDIA '95)*. San Francisco: ACM Press.

DRUSCHEL, PETER, ABBOTT, MARK B., PAGELS, MICHAEL, & PETERSON, LARRY L. 1993. Network Subsystem Design. *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, **7**(4), 8–17.

FONSECA, H. 1994 (June). *Ambientes de Suporte para a Modularização, Concretização e Execução de Protocolos de Comunicação*. M.Phil. thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa.

GAMMA, ERICH, HELM, RICHARD, JOHNSON, RALPH, & VLISSIDES, JOHN. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GARBINATO, BENOÎT, & GUERRAOUI, RACHID. 1997. Using the Strategy Design Pattern to Compose Reliable Distributed Protocols. *Pages 221–232 of:* USENIX (ed), *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 16–19, 1997. Portland, Oregon*. Berkeley, CA, USA: USENIX.

GARBINATO, BENOÎT, FELBER, PASCAL, & GUERRAOUI, RACHID. 1996. Protocol Classes for Designing Reliable Distributed Environments. *In: European Conference on Object-Oriented Programming Proceedings (ECOOP'96)*. Lectures in Computer Science, vol. 1098. Linz, Austria: Springer Verlag.

HAYDEN, M. 1998. *The Ensemble System*. Ph.D. thesis, Cornell University, Computer Science Department.

HÜNI, HERMANN, JOHNSON, RALPH E., & ENGEL, ROBERT. 1995 (Oct.15–19). A Framework for Network Protocol Software. *Pages 358–369 of: Proceedings of OOPSLA'95, Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*.

HUTCHINSON, N., & PETERSON, L. 1991. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, **17**(1), 64–76.

ISO. 1982. Reference Manual of Open Systems Interconnection - OSI. *ISO 7498*.

MIRANDA, H., & RODRIGUES, L. 1999a (Apr.). Communication support for multiple QoS requirements. *In: Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*.

MIRANDA, H., & RODRIGUES, L. 1999b (Sept.). Flexible Communication Support for CSCW Applications. *Pages 338–342 of: 5th Internation Workshop on Groupware - CRIWG'99*. IEEE, Cancún, México.

MIRANDA, H., ANTUNES, M., RODRIGUES, L., & RITO SILVA, A. 2000a (Oct.). Group Communication Support for Dependable Multi-User Object Oriented Environments. *In: Proceedings of the International SRDS Workshop on Dependable System Middleware and Group Communication (DSMGC 2000), in conjunction with the IEEE Symposium on Reliable Distributed Systems (SRDS'19)*.

MIRANDA, HUGO, COSTA, FRANCISCO, & RODRIGUES, LUÍS. 1998 (Nov. 9–10). Realização de um serviço de grupos ligeiros para a plataforma de comunicação em grupo Ensemble. *Pages 65–69 of: Actas da 1ª conferência sobre Redes de Computadores*. Universidade de Coimbra e Fundação para a Computação Científica Nacional.

MIRANDA, HUGO, PINTO, ALEXANDRE, & RODRIGUES, LUÍS. 2000b (Nov.). *Application Program Interface Specification of Appia v1.1*. Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa.

MIRANDA, HUGO, PINTO, ALEXANDRE, & RODRIGUES, LUÍS. 2001. Appia, a flexible protocol kernel supporting multiple coordinated channels. *Page to appear of: Proceedings of The 21st International Conference on Distributed Computing Systems (ICDCS-21)*. Phoenix, Arizona, USA: IEEE Computer Society.

MISHRA, S., & YANG, R. 1998. Thread-based vs. Event-based Implementation of a Group Communication Service. *Pages 398–402 of: Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*. Orlando, Florida, USA: IEEE Computer Society.

MOORE, GORDON E. 1965. Cramming More Components Onto Integrated Circuits. *Electronics*, **38**(8).

MOSBERGER, DAVID. 1996 (Jan.). *Message Library Design Notes*.

O'MALLEY, S., & PETERSON, L. 1992. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, **10**(2), 110–143.

PEREIRA, JOSÉ ORLANDO, & OLIVEIRA, RUI. 1997 (Oct.). Object-Oriented Open Implementation of Reliable Communication Protocols. *In: Workshop on Dependable Distributed Object Systems, OOPSLA'97*.

PLUMMER, DAVID C. 1982 (Nov.). *An Ethernet Address Resolution Protocol*. Request For Comments 826. Network Working Group. ARP.

POSTEL, J. 1980 (Aug.). *User Datagram Protocol*. Request for Comments 768. USc Inf. S. Inst.

POSTEL, J. 1981a (Sept.). *Internet Protocol*. Request for Comments 791. USc Inf. S. Inst.

POSTEL, J. 1981b (Sept.). *Transmission Control Protocol*. Request for Comments 793. USc Inf. S. Inst.

RHEE, I., CHEUNG, S., HUTTO, P., & SUNDERAM, V. 1997 (May). Group Communication Support for Distributed Collaboration Systems. *Pages 43–50 of: Proceedings of the 17th International Conference on Distributed Computing Systems*. IEEE, Balitmore, Maryland, USA.

RITCHIE, D. M. 1984. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, **63**(8, Part 2), 1897–1910.

RODRIGUES, J., MIRANDA, H., VENTURA, J., & RODRIGUES, L. 2001. The design of RTAppia. *Page to appear of: Proceedings of the Sixth IEEE International Workshop on Object-oriented Real-Time Dependable Systems*. Rome, Italy: IEEE.

RODRIGUES, L., GUO, K., SARGENTO, A., VAN RENESSE, R., GLADE, B., VERÍSSIMO, P., & BIRMAN, K. 1996 (Oct.). A Transparent Light-Weight Group Service. *Pages 130–139 of: Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*.

SILBERCHATZ, ABRAHAM, GALVIN, PETER, & GAGNE, GREG. 2000. *Applied Operating System Concepts*. first edn. John Wiley & Sons, Inc.

TANENBAUM, ANDREW S. 1996. *Computer Networks*. Third international edn. Upper Saddle River, New Jersey 07458: Prentice-Hall International, Inc.

TENNENHOUSE, DAVID L. 1989. Layered Multiplexing Considered Harmful. *In:* RUDIN, & WILLIAMSON (eds), *Protocols for High-Speed Networks*. Amsterdam: North Holland.

VAN RENESSE, R., HICKEY, T., & BIRMAN, K. 1994 (Aug.). *Design and Performance of Horus: A Lightweight Group Communications System*. Tech. rept. 94-1442. Department of Computer Science, Cornell University, Ithaca, NY 14853-7501.

VAN RENESSE, R., BIRMAN, KEN, & MAFFEIS, S. 1996. Horus: A Flexible Group Communications System. *Communications of the ACM*, **39**(4), 76–83.

VAN RENESSE, ROBBERT, BIRMAN, KENNETH P., FRIEDMAN, ROY, HAYDEN, MARK, & KARR, DAVID A. 1995a (2–23 Aug.). A Framework for Protocol Composition in Horus. *Pages 80–89 of: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*.

VAN RENESSE, ROBBERT, BIRMAN, KENNETH P., GLADE, BRADFORD B., GUO, KATIE, HAYDEN, MARK, HICKEY, TAKAKO, MALKI, DALIA, VAYSBURD, ALEX, & VOGELS, WERNER. 1995b (Mar. 23,). *Horus: A Flexible Group Communications System*. Technical Report TR95-1500. Cornell University, Computer Science Department.

VAN RENESSE, ROBBERT, BIRMAN, KEN, HAYDEN, MARK, VAYSBURD, ALEXEY, & KARR, DAVID. 1997 (July). *Building Adaptive Systems Using Ensemble*. Tech. rept. TR97-1638. Cornell University.

X-KERNEL. 1997 (June). *x-Kernel Programmer's Manual (version 3.3)*. Network Systems Research Group.

Índice Remissivo

I

J

L

M

N

O

P