Modelo de Referência Unificado para Arquitetura de Protocolos e Programação de Aplicações Multimídia*

Sérgio Colcher colcher@inf.puc-rio.br
Luiz F. G. Soares lfgs@inf.puc-rio.br

Laboratório TeleMídia
Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática – PUC-Rio
R. Marquês de São Vicente, 225
Rio de Janeiro – RJ
22453-900, Brazil
Tel.: +55 (021) 529-9461
Fax.: +55 (021) 259-2232

Resumo

Reconhecendo que a tecnologia de orientação por objetos (OO) é eficaz tanto na modelagem e implementação de aplicações distribuídas como na de componentes de protocolos de comunicação, o presente trabalho propõe um novo modelo de referência a ser utilizado no projeto de sistemas distribuídos, que permite a construção e configuração das aplicações em ambientes flexíveis no que diz respeito à adequação dos serviços de comunicação às características específicas de tráfego e QoS das aplicações. O modelo de referência é composto de (i) um modelo de OO, (ii) um conjunto de design patterns e (iii) um modelo de ciclo de vida de serviços. Neste artigo, apenas os dois primeiros serão abordados. O modelo é genérico, podendo ser aplicado na modelagem de camadas de comunicação arbitrárias em arquiteturas quaisquer, sendo referenciado como um meta-modelo para construção de arquiteturas distribuídas.

Abstract

Realizing that OO techniques are well-suited to develop the end-system applications as well as to model, implement, configure and reconfigure services within communication networks, this work presents a new reference model that may be applied to the design of distributed systems that are flexible with respect to the adaptability of communication services to specific requirements of applications. The reference model is composed of (i) an OO model, (ii) a set of design patterns and (iii) a service life-cycle. Only the first and second items are addressed in this paper. The model is generic and may be used to model specific any communication layer within arbitrary architectures, being referred as a meta-model for the construction of distributed architectures.

1 Introdução

Nos últimos anos, tem-se observado uma crescente demanda por serviços de telecomunicações que ofereçam suporte ao transporte de tipos de tráfego com características bastante diversificadas e com requisitos de qualidade de serviço (Quality of Service – QoS) que variam muito de acordo com as aplicações. As origens dessa crescente demanda podem ser

*Este trabalho foi parcialmente financiado pela Empresa Brasileira de Telecomunicações (Embratel).

atribuídas a dois fatores: (i) a disseminação, a baixo custo, de dispositivos e técnicas de processamento, compressão, codificação e tratamento das diferentes mídias [22], e (ii) o desenvolvimento da tecnologia de redes de alta velocidade e integração de serviços, que criou uma expectativa quanto aos novos serviços de banda larga e aplicações multimídia distribuídas que poderão ser desenvolvidos [23].

Se, por um lado, tecnologias como a das redes ATM oferecem algumas soluções para a realização de redes com integração de serviços e garantias de QoS, quando procura-se implementar serviços específicos sobre essas redes, a variedade de requisitos torna-se um obstáculo. Estratégias, como organizar serviços de comunicação em classes (com características comuns), procuram diminuir tais dificuldades, mas ainda deixam a desejar, especialmente quando procura-se desenvolver um único protocolo (ou um número finito de protocolos) que atenda a todas as classes de serviços e possíveis QoSs — combinações que tendem a ser cada vez mais numerosas.

Uma alternativa considerada em recentes propostas como [15, 18, 24, 10], e que também será explorada no presente trabalho, é a de oferecer um serviço de comunicação configurável, que possa ser adaptado de acordo com a necessidade das aplicações finais. De uma forma geral, a principal característica desses trabalhos é a utilização de uma modelagem orientada a objetos (OO) para a definição de pequenos componentes de protocolo reutilizáveis. Através da interconexão desses componentes é possível construir implementações de protocolos cujas estruturas são descritas através do grafo da interligação dos seus componentes, denominados *grafos de protocolos*. Unidades de informação trafegam através desses componentes obedecendo a organização do grafo e, a cada nó, uma parte do processamento é efetuado. Quando adequadamente projetados, os componentes podem servir como base para a configuração dinâmica dos protocolos, através da configuração da própria estrutura do grafo (que pode permitir adições, remoções ou substituições dos seus componentes).

Uma consequência direta do modelo acima é que, como os diferentes requisitos são oriundos de aplicações ou de serviços de alto nível, parte da responsabilidade de implementação e, principalmente, configuração dos sistemas de comunicação recairá sobre os projetistas, analistas e programadores das próprias aplicações. Essa "programabilidade", conforme mencionada por Lazar em [10], deverá ser uma característica chave em qualquer ambiente distribuído moderno. O projeto, implementação e configuração de serviços de comunicação deverá se tornar tão corriqueiro quanto o projeto da aplicação em si, estando ambas as tarefas interligadas e completamente integradas.

Embora a tecnologia de OO seja reconhecidamente eficaz na modelagem e implementação tanto das aplicações distribuídas, residentes nos equipamentos dos usuários, como dos componentes de protocolos de comunicação, que podem residir em qualquer equipamento distribuído pela rede, ainda não se encontram trabalhos que tratem detalhadamente de um modelo integrado de utilização dessa tecnologia para ambos os campos de atuação (programação de aplicações e implementação/configuração de protocolos) e, principalmente, que levem em consideração sistemas com requisitos de QoS.

O presente trabalho propõe um modelo de referência a ser utilizado no projeto de sistemas multimídia distribuídos, que permite a construção e configuração de ambientes flexíveis, no que diz respeito a adequação dos serviços de comunicação às características específicas de tráfego e QoS das aplicações. O modelo inclui *um modelo de objetos* que incorpora características para tratamento de tipos contínuos e QoS, *um conjunto de design patterns* que permite a utilização do modelo em diferentes escalas de distribuição e concorrência dos objetos, e um *modelo de ciclo de vida de serviços*, que rege o estado dos componentes do ambiente de forma a permitir a configuração em qualquer fase de sua existência.

O modelo de objetos é dividido em duas partes. A primeira, a *visão computacional*, corresponde a uma especificação completamente independente de aspectos de distribuição e concorrência dos objetos, sendo a chave para a integração dos vários níveis de abstração nos quais o modelo como um todo pode ser aplicado. A visão computacional traz algumas inovações em relação a modelos tradicionais, tratando objetos como máquinas virtuais e generalizando vários conceitos definidos em trabalhos como [16, 9, 24]. A segunda, a *visão de engenharia*, é baseada numa sequência de refinamentos, cada qual relacionado com um nível diferente de distribuição e concorrência dos objetos, que modifica e estende amplamente as noções básicas da visão de engenharia definida pelo RM-ODP [9]. Nesses refinamentos, inclui-se também um design pattern específico associado a cada nível definido, que serve como guia para a construção da infra-estrutura do nível subsequente.

O modelo de ciclo de vida dos serviços, cuja função é regular a operação dos componentes do sistema de forma a evitar que inconsistências ocorram devido à falta de cooperação entre os processos de configuração, instalação, operação, etc., não será tratado neste artigo. Apenas como exemplo de seu papel, através do modelo de ciclo de vida pode-se estabelecer que a substituição de um determinado componente de uma entidade de protocolo poderá ter de ser precedida pela suspensão temporária da operação de um serviço de mais alto nível.

Embora motivado pela variedade de requisitos presente em sistemas multimídia e redes com integração de serviços, o modelo apresentado é genérico e aplicável a ambientes distribuídos genéricos. Adicionalmente, o modelo é independente da arquitetura da rede e dos níveis de protocolo, podendo ser classificado como um *meta-modelo para construção de arquiteturas distribuídas*. O produto de sua utilização por projetistas e programadores é a definição de entidades, camadas e arquiteturas, acompanhada de uma implementação que garantirá o grau de flexibilidade e capacidade de configuração desejados.

O artigo encontra-se estruturado da seguinte forma. Inicialmente, dedica-se a Seção 2 à apresentação de um resumo das principais características de trabalhos considerados significativos no tratamento de modelos de referência, design patterns, programação, serviços e arquiteturas com tratamento de QoS. Procura-se estabelecer um relacionamento geral entre esses vários trabalhos e ressaltar os aspectos sobre os quais o modelo proposto é completamente inovador. Em seguida, na Seção 3, passa-se à apresentação do modelo de objetos, sua visão computacional e de engenharia. Finalizando, na Seção 4, apresentam-se algumas conclusões que ressaltam os aspectos nos quais o presente trabalho oferece soluções e os que ainda necessitam ser aprofundados.

2 Trabalhos Relacionados

Ao se falar de modelos de referência, o *OSI-RM* [8] é, certamente, obrigatório. As regras gerais de divisão em camadas do OSI-RM serão tratadas, no presente trabalho, de acordo com uma visão OO, levando em consideração vários fatores referentes a composição e estruturação de entidades de protocolo configuráveis. Adicionalmente, trata-se de aprofundar a visão do ambiente de execução local (LSE — Local System Environment) de cada subsistema (conforme nomenclatura do próprio OSI). Em particular, a definição de um ambiente de processamento padronizado é crucial para permitir a configuração dinâmica de componentes de protocolo. Ambientes de execução serão baseados na definição de um *mediador*, uma entidade abstrata para definição de um ambiente genérico para execução e troca de mensagens entre objetos.

Pode-se atribuir a idéia geral de mediadores entre objetos à definição do *CORBA (Common Object Request Broker Architecture)* [16]. A principal idéia do CORBA é definir um conjunto de interfaces a mecanismos que permitam a interação entre objetos de

forma independente da sua distribuição e das linguagens de programação utilizadas na sua codificação. Essa transparência é obtida através da atuação de um elemento central denominado ORB (Object Request Broker), representado pela seta grande da Figura 1. O núcleo do ORB (ORB core) é a parte responsável pela comunicação entre os objetos.

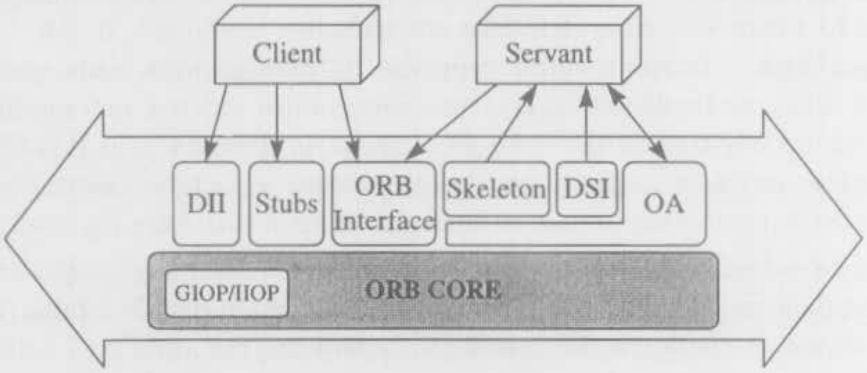

Figura 1: Componentes do CORBA.

Stubs e skeletons são os intermediários dessa comunicação que, de um lado fornecem aos objetos (cliente e servidor) a abstração de que estão se comunicando com objetos locais, em sua linguagem de programação nativa; do outro lado, stubs e skeletons comunicam-se com o núcleo do ORB, transformando chamadas a operações em mensagens de solicitação e resposta e entregando-as ao ORB para que sejam encaminhadas e retornadas. Auxiliando o núcleo, outros componentes permitem que os objetos sejam ativados, encontrados e corretamete disparados, dos quais ressalta-se o Object Adapter (OA). Ao OA cabem todas as funções de ativação do objeto servidor, incluindo as funções de escalonamento, que serão mencionadas ao longo do presente trabalho, em vários níveis da visão de engenharia do modelo de objetos. Objetos interagem com OAs da mesma forma como com qualquer outro objeto, apesar de OAs serem internos ao ORB. Objetos que apresentam essa característica são referenciados pelo CORBA como pseudo-objetos.

O mediador, segundo proposto neste artigo, corresponde a um intermediário, que pode ser tão simples quanto o próprio ambiente de execução de uma linguagem OO, ou tão complexo quanto um ORB definido pelo CORBA. A estrutura genérica ORB definida pelo CORBA correponderá ao mediador de mais alto nível do modelo, que é estendido de forma a prover o tratamento a tipos de dados contínuos. Os pipes e MediaPipes em cada nível, correspoderão a pseudo-objetos.

Ainda com relação a modelos de referência, o *Modelo de Referência para Processamento Aberto e Distribuído (Reference Model of Open Distributed Processing — RM-ODP)* [9] é um esforço conjunto dos órgãos internacionais de padronização ISO e ITU-T para o desenvolvimento de uma estrutura básica que pode ser utilizada na especificação de sistemas abertos. O RM-ODP define que a especificação de um sistema aberto deve ser dividida em cinco visões, e define um conjunto de conceitos para cada visão, denominada "linguagem daquela visão", que permitem essas especificações. Os conceitos envolvidos nas linguagens baseiam-se, todos, num modelo de orientação a objetos.

Na presente proposta, concentrou-se a atenção em duas visões que foram inspiradas em duas das cinco visões do RM-ODP: a *visão computacional* e a *visão de engenharia*. O objetivo da divisão em visões é, basicamente o mesmo do RM-ODP, porém a estratégia adotada e, em particular, a linguagem da visão de engenharia é sensívelmente mais aprofundada no presente trabalho, sendo os patterns para composição de protocolos bastante mais detalhados do que as regras gerais do RM-ODP.

Em relação aos design patterns utilizados, em uma descrição mais global, poderia-

se classificá-los como patterns ao estilo *Pipes e Filtros* [13, 1], no qual uma atividade é dividida em várias sub-atividades independentes, processadas sequencialmente ou em paralelo de acordo com algum grafo de precedências. A concepção de pipes e filtros adapta-se perfeitamente a construção de protocolos, e alguns trabalhos já exploraram essas idéias como o *Conduits+* [5, 15] e o *x-Kernel* [6]. Nesse artigo, apresentam-se algumas extensão a essas idéias que incluem a definição de pipes especiais — os MediaPipes — e a utilização da idéia do mediador como hospedeiro dos pipes, de forma que a definição de pipes permanece abstrata, enquanto a implementação pode ser dependente do tipo de distribuição e concorrência de uma determinada plataforma.

O *x-Kernel*[6] é o núcleo de um sistema operacional que fornece uma arquitetura para a construção de protocolos de rede eficientes. Sua principal característica está no tratamento da sincronização entre camadas e passagem de mensagens minimizando o número de cópias de memória. O presente trabalho inspira-se no pattern para tratamento de buffers do x-Kernel, estendendo-o para o tratamento de mensagens tipadas.

Afastando-se da idéia de componentes homogêneos (filtros) interligados por pipes, o *Adaptive Communication Environment (ACE)* [17] implementa um conjunto de design patterns cujo principal objetivo é simplificar os mecanismos de comunicação entre processos oferecidos por sistemas operacionais. Adicionalmente, o ACE procura automatizar a configuração e reconfiguração do software de comunicação. A definição de arquitetura de processo como forma de classificar o paralelismo na construção de arquiteturas de protocolos é uma das influências do ACE no presente trabalho. Alem disso, o ACE foi utilizado na construção de um ORB multi-threaded, denominado *TAO (The ACE ORB)* [21, 19, 20], que permite a especificação de requisitos de QoS baseados em parâmetros oriundos da aplicação. Baseado nessas especificações o TAO é capaz de escalonar os recursos como CPUs, memória, banda passante da rede, etc., mecanismos que encontram-se intimamente relacionados com o modelo proposto no presente trabalho. Não há porém, no ACE ou no TAO, um esforço no sentido de consolidar os conceitos de OO, estando esses trabalhos mais voltados para os aspectos práticos de implementação e compatibilização entre os sistemas operacionais utilizados.

3 Modelo de Objetos

O foco deste artigo, em relação ao modelo de objetos, encontra-se na definição dos aspectos relacionados ao modelo de execução de um ambiente OO mais diretamente ligados à interação e comunicação entre objetos em tempo de execução. Evitou-se o aprofundamento maior sobre questões do modelo de construção, mais especificamente, os de definição de tipos e subtipos e suas variantes em relação à verificação estática e dinâmica.

O modelo encontra-se dividido em duas partes. A primeira corresponde a uma especificação completamente independente de aspectos de distribuição e concorrência dos objetos, e pode ser associada, segundo a nomenclatura utilizada pelo RM-ODP, aos conceitos básicos de uma linguagem para a visão computacional. A segunda é baseada em uma sequência de refinamentos, cada qual relacionado com um nível diferente de distribuição e concorrência dos objetos, podendo ser associada aos conceitos básicos de uma linguagem para a visão de engenharia do RM-ODP. Cada um dos níveis da visão de engenharia está relacionado, sucessivamente, a uma diferente escala na separação entre os objetos, que podem estar numa mesma thread, em threads diferentes de um mesmo processo ou em threads diferentes de processos arbitrários.

3.1 Visão Computacional

O modelo de objetos na visão computacional é a chave para a integração dos vários níveis de abstração aos quais o modelo como um todo pode ser aplicado. A visão computacional oferece uma descrição abstrata de alto nível, válida em quaisquer dos níveis a serem definidos. A visão computacional pode ser descrita com o auxílio da Figura 2.

Figura 2: Componentes da Visão Computacional.

Clientes e servidores correspondem a *objetos*. Objetos são definidos como *máquinas virtuais,* capazes de executar instruções. Cada objeto possui identidade e ciclo de vida próprios, e poderá conter, de acordo com sua definição, um conjunto de variáveis internas, que determinam o seu *estado* a qualquer instante de tempo.

Uma instrução é composta de sua identificação e possíveis parâmetros. O efeito da execução de uma instrução por um objeto, que depende do estado do objeto e dos parâmetros da instrução, poderá ser a alteração de seu estado e o envio de mensagens a outros objetos.

Objetos interagem através de trocas de mensagens, que constituem o mecanismo central de todo o ambiente de execução definido pela visão computacional. Mensagens se prestam a várias funções dentro do modelo. Para executar instruções, objetos são acionados através de mensagens recebidas, que contém uma ou mais instruções, e são endereçadas a um objeto determinado. Mensagens também serão utilizadas, em determinados casos, para transportar valores.

Uma *operação* é definida como sendo uma composição de: (i) uma mensagem (denominada *solicitação*) enviada por um objeto (denominado cliente da operação) a um outro objeto (denominado servidor da operação) contendo uma e somente uma instrução; (ii) uma possível mensagem de *resposta* do servidor ao cliente, contendo um ou mais valores. Dois tipos de operação são possíveis: operações *síncronas* e operações *assíncronas*, porém, em determinados níveis definidos pela visão de engenharia, apenas uma das duas será possível. A definição de uma operação síncrona é a de uma operação na qual o cliente é impedido de prosseguir em sua atividade até que a resposta do servidor seja recebida. Em uma operação assíncrona, por outro lado, o cliente envia a solicitação e continua sua atividade. Em uma operação assíncrona não há mensagem de resposta, enquanto que em uma operação síncrona sempre há uma mensagem de resposta, mesmo que não haja resultados específicos a serem retornados.

De forma análoga, uma *tarefa* é definida como sendo uma composição de (i) uma mensagem (denominada *solicitação*) enviada por um objeto (denominado cliente da tarefa) a um outro objeto (denominado servidor da tarefa) contendo um conjunto de instruções; (ii) uma possível mensagem de *resposta* do servidor ao cliente, contendo um ou mais valores. Assim como operações, tarefas podem ser síncronas ou assíncronas. Uma operação é uma tarefa na qual a solicitação é composta por apenas uma única instrução.

Mensagens são trocadas pelos objetos através de uma entidade abstrata denominada de *mediador*. Considerando o caráter abstrato do modelo, trocas de mensagens são processos puramente conceituais, que poderão ser materializados por diferentes mecanismos, dependendo do nível da visão de engenharia considerado, sendo o mediador a entidade responsável por essa materialização. Define-se *ambiente de processamento virtual* como sendo composto de um mediador e as máquinas virtuais (ou objetos) que o utilizam.

O mediador de um ambiente de processamento virtual é composto de um *núcleo*, que permite a troca de mensagens entre objetos, e alguns objetos, internos ao próprio mediador, que juntamente com o núcleo, oferecem um conjunto de serviços básicos. Esses serviços básicos podem, dessa forma, ser obtidos por objetos clientes como se fossem serviços oferecidos por um servidor comum, acima do mediador. Assim, pode-se fazer uma analogia entre serviços básicos e pseudo-objetos definidos pelo CORBA.

O mais importante pseudo-objeto definido pelo mediador é o *pipe*. Um pipe corresponde a uma associação explícita entre objetos para troca de informação ao estilo "memória compartilhada". O estabelecimento de um pipe e sua utilização é uma interação entre os objetos e o mediador.

Considerando a necessidade de exercer controles sobre a qualidade de trocas de mensagens específicas, em especial quando mensagens estão associadas a operações ou transporte de dados multimídia, o mediador oferece suporte à definição de um pipe especial denominado *MediaPipe* [2]. MediaPipes podem ser estabelecidos entre cada conjunto de objetos sobre o qual se deseja exercer controle. No processo de estabelecimento, um conjunto de valores que representam o tráfego e os requisitos de QoS são fornecidos. A partir daí é tarefa do mediador: (i) aceitar ou não a criação do referido MediaPipe e (ii) caso aceite, garantir as características requisitadas. O conjunto de valores bem como os mecanismos utilizados para garantir essas características são totalmente dependentes do nível de distribuição e concorrência entre os objetos e, portanto, assuntos da visão de engenharia.

3.2 Visão de Engenharia

O objetivo da visão de engenharia do presente modelo é, de forma geral, semelhante ao objetivo determinado pelo RM-ODP. A definição, porém, é bastante distinta. No modelo proposto neste trabalho, a visão de engenharia relaciona um determinado ambiente de processamento virtual a recursos computacionais disponíveis em um possível sistema de processamento real. Recursos de um sistema de processamento real incluem as noções de thread, processo, máquina e rede. Em relação à nomenclatura, apesar de coincidente em vários pontos (como cluster, cápsula e nó), não há uma correspondência direta entre conceitos aqui utilizados e os definidos pelo RM-ODP, apesar de que algumas semelhanças podem ser apontadas ao longo da presente descrição.

3.2.1 Cluster e Cápsula

Um determinado ambiente de processamento virtual é dito um *cluster*, quando as atividades de todos os objetos e do mediador ocorrem todas, sempre, numa mesma thread. Assim, clusters possuem uma capacidade de processamento referente a uma parcela do processamento de um processador real. Um ambiente de processamento virtual é dito uma *cápsula*, quando as atividades de todos os objetos e do mediador ocorrem todas, sempre, num mesmo processo. Uma cápsula tem um espaço de endereçamento único e uma ou mais threads de execução.

Dentro da concepção de um modelo integrador para as atividades de (i) programação de aplicações multimídia distribuídas e (ii) confecção de protocolos e serviços de comuni-

cação configuráveis, clusters e cápsulas são, tipicamente, ambientes de suporte ao segundo ítem, já que tratam-se de modelos que consideram interações dentro de um mesmo processo e, portanto deverão ser aproveitados em patterns que tratem da *comunicação vertical* dentro de um mesmo subsistema (segundo nomenclatura do modelo OSI – Seção 2).

Patterns para Comunicação Vertical

A forma geral dos patterns para a construção de protocolos de comunicação é a de um grafo, no qual cada nó corresponde a um componente de protocolo, e as arestas a interações entre componentes por onde unidades de informação do protocolo passam como mensagens entre objetos. A Figura 3 exemplifica um fluxo de chegada de unidades de informação em um grafo de protocolos e as interações entre os objetos através dos pipes.

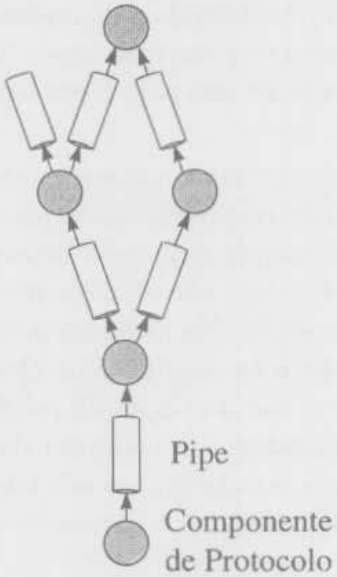

Figura 3: Pipes e Componentes de Protocolo em um Fluxo de Chegada.

As passagens das unidades de informação ao nível de cluster e cápsula são realizados por intermédio de buffers entre os componentes, que juntamente com os mecanismos do mediador que permitem a passagem de mensagens entre objetos, correspondem aos pipes.

Componentes de protocolo podem ser estruturados, internamente, em outros grafos de protocolos, formados, novamente, por componentes, de granularidade mais fina, e pipes. Esse tipo de estruturação permite a organização lógica e hierárquica da função de protocolos. Define-se um *componente simples* como sendo um componente não estruturado. Já um *componente composto* é formado por outros componentes e pipes.

Buffers de Mensagens Tipadas

Componentes de protocolo retiram unidades de informação de pipes de entrada, processam-nas, eventualmente as armazenam temporariamente e, finalmente, colocam novas unidades de informação em pipes de saída. Nesse nível, assim como em outros, uma das chaves para eficiência do protocolo é a redução do número de cópias ou movimentações de dados entre buffers. A estratégia básica para essa diminuição está em trabalhar com buffers que manipulam apenas referências a unidades de informação (que chamaremos, genericamente, de mensagens) que estão armazenadas em um buffer único. A passagem das mensagens de um buffer para outro passa a ser uma operação de manipulação de ponteiros, ao invés de cópias entre regiões de memória (que consomem altos tempos de CPU). A passagem

de mensagens entre buffers não é o único problema relativo a mensagens nesse nível. Outras duas questões podem ser apontadas, uma relativa à fragmentação das mensagens nos buffers, e outra ao acesso organizado aos campos de uma mensagem.

A primeira origina-se da manipulação das mensagens pelos componentes de protocolo, que podem, eventualmente, adicionar ou remover campos. No processo de remoção de campos, mensagens deixam espaços no buffer que as contém, os quais dificilmente serão reutilizados. Já na adição de campos, uma nova área para o campo a ser inserido terá de ser alocada em uma região possivelmente não contígua à área que a mensagem já ocupa. A tradicional solução de alocar uma nova região, maior que a original e suficientemente grande para acomodar a nova mensagem inteira, deve ser obviamente evitada, pois incorre em cópias de regiões de memória.

A segunda questão diz respeito ao acesso ordenado aos campos de uma mensagem. Unidades de informação de protocolo são normalmente estruturadas em campos que, por sua vez, podem ser sucessivamente estruturados em outros campos. Dessa forma, mensagens podem ser associadas a tipos, cuja estrutura interna pode ser encapsulada através de operações que fornecem os mecanismos para a sua manipulação. Como exemplo, considere o caso de um componente de protocolo que manipula mensagens cuja estrutura contém um cabeçalho que, por sua vez, contém um campo com um endereço de destino. O componente em questão pode ter acesso a esse endereço através de uma operação aplicada sobre a mensagem (algo como `get_destAddr()`) que retorna um valor de um tipo definido (como `address`, por exemplo).

Em relação às questões de passagem de mensagens e alocação de regiões não contíguas, já encontram-se, na literatura, modelos para o tratamento de mensagens encapsuladas em abstrações de buffer contínuo e manipulação de referências, como o apresentado pelo x-Kernel [14, 6] (mencionado na Seção 2). Por outro lado, pouco se encontra em termos de uma definição genérica para o tratamento do acesso ordenado a campos de uma mensagem. Em geral, deixa-se a responsabilidade desse acesso à própria implementação dos componentes de protocolo, que deverão, a partir da inspeção das posições ocupadas pela mensagem, descobrir os campos e convertê-los aos tipos desejados. Implementado dessa forma, o acesso a campos de mensagens é um mecanismo desprovido de tratamentos ou verificações estáticas de tipos, que poderiam detectar, a priori, usos ou acessos indevidos.

A seguir, apresenta-se um modelo que integra ambos os mecanismos em um pattern único, denominado *design pattern para buffers de mensagens tipadas*. Sua utilização será, tipicamente, na implementação de pipes.

Mensagens são armazenadas em um buffer de bytes único, no qual elas ocuparão, possivelmente, ao longo do tempo, regiões não contíguas. Tais mensagens são encapsuladas por uma classe de objetos que fornece uma abstração de que seus bytes encontram-se posicionados sequencialmente, de forma contígua. O mecanismo que permite tal abstração é baseado numa estrutura em árvore, cujas folhas são referências a regiões contíguas no buffer de bytes, de forma semelhante ao definido em [14].

Utilizando-se a abstração de mensagens contíguas, define-se a abstração de *mensagem tipada*, que apresenta uma interface com operações de acesso aos campos associados a tipos definidos. A implementação de mensagens tipadas é baseada na descrição do intervalo de bytes ocupados por cada campo, que pode, inclusive, depender de valores que outros campos assumem.

Mensagens tipadas são manipuladas através de *buffers de mensagens tipadas*. Tais buffers são associados a cada pipe e, eventualmente, a componentes de protocolos, no momento de sua configuração. Buffers de mensagens tipadas oferecem uma interface com operações que permitem a colocação e retirada de mensagens. A implementação da

operação de colocação é responsável por inspecionar a mensagem e atualizar os intervalos de cada campo de forma a fazê-los refletir a estruturação da mensagem.

Como exemplo de utilização, considere um componente de protocolo com um pipe de entrada e um de saída. Considere, ainda, um fluxo de chegada, no qual mensagens do pipe de entrada tem um cabeçalho e um fecho que são processados e retirados pelo componente de protocolo, além de uma parte de dados (cuja estrutura interna é desconhecida). Assim, ao invocar a operação de colocação de mensagens no pipe de saída, as novas mensagens são automaticamente atualizadas de forma a revelar uma estrutura interna, desconhecida pelo componente anterior, com um novo cabeçalho e fecho, que agora podem ser utilizados no processamento de um componente subsequente.

Componentes de Protocolo

Componentes de protocolo, sejam compostos ou simples, podem ter várias entradas e saídas, conforme ilustrado na Figura 3. Entradas e saídas são univocamente identificadas, em cada componente, por valores pertencente a um tipo denominado *endereço de e/s*. No processo de configuração, esses identificadores serão utilizados pelos componentes compostos para permitir a exata representação da topologia de seu grafo de protocolo (quais pipes estão conectados a quais componentes e em que endereço de e/s). Uma operação `bind(pipeRef,IOAddr)`, que permite associar um pipe (referenciado por `pipeRef`) a uma entrada ou saída (identificada pelo endereço `IOAddr`), deve ser oferecida por todos os componentes de forma que cada um tenha como receber e armazenar referências aos seus pipes de entrada ou saída, possibilitando a colocação ou retirada de mensagens nesses pipes. Assim, o processo de configuração cria componentes, cria pipes, coloca-os em composições e interliga-os através da passagem das referências dos pipes aos respectivos componentes, utilizando a operação `bind`.

Em componentes simples, é responsabilidade de quem implementa definir de que forma as várias entradas serão atendidas. Em composições, alguns dos componentes internos devem estar diretamente associados a entradas ou saídas do componente composto. Tais componentes são denominados *portas*. Apenas uma porta pode estar associada a uma entrada ou saída, mas uma mesma porta pode estar associada a mais de uma entrada ou saída, conforme ilustrado na Figura 4 (portas correspondem aos componentes acinzentados).

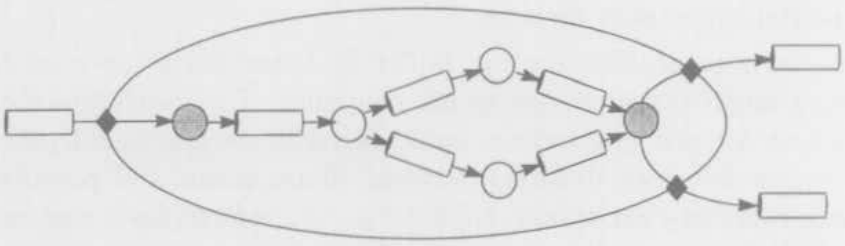

Figura 4: Associação entre Endereços de E/S de um Componente Composto e Endereços de E/S das Portas.

Cada componente composto mantém um conjunto de tuplas do tipo `(IOAddr1, (portRef, IOAddr2))`, onde `IOAddr1` é o endereço de e/s do componente composto, `portRef` é uma referência a uma porta e `IOAddr2` é o endereço de e/s nessa porta. Componentes compostos implementam operações que permitem recuperar, através de consultas a essas tuplas, o par `(portRef, IOAddr2)`, a partir de `IOAddr1`. Quando uma operação de configuração `bind(pipeRef, IOAddr1)` é solicitada a um componente composto, sua implementação,

automaticamente, invoca a operação `portRef.bind(pipeRef, IOAddr2)`. Assim, os verdadeiros responsáveis pelos acessos aos pipes ligados a um componente composto são suas portas. Se as portas de um componente composto também forem componentes compostos, as portas dessas portas é que serão as responsáveis, e assim sucessivamente, até que uma porta seja um componente simples.

Cada componente, simples ou composto, implementa uma operação `run()` que, quando solicitada, provoca a execução de suas funções, que incluem: a inspeção de pipes de entrada, a retirada de mensagens de um ou mais desses pipes, o processamento propriamente dito e a colocação de mensagens em um ou mais pipes de saída. A implementação de `run()` em componentes simples é total responsabilidade de programadores e projetistas do sistema. Em componentes compostos, a implementação de `run()` segue padrões bem definidos, que diferem no nível de cápsula para o nível cluster, e serão, portanto, tratados separadamente, de forma detalhada, nas seções relativas aos patterns específicos de cluster e cápsula. De uma maneira geral, a implementação de `run()`, em ambos os casos, irá disparar a operação `run()` em cada um de seus componentes.

O pattern para comunicação vertical define uma regra geral para alguns níveis de composição que devem ser encontrados numa arquitetura qualquer: sempre devem existir componentes que delimitem exatamente as fronteiras de entidades de protocolo de cada camada e subcamada definida pela arquitetura de comunicação. Tais componentes (simples ou compostos) são denominados *objetos entidade* (ou simplesmente *entidades*). Internamente aos objetos entidade, composições são livres. Já as composições feitas a partir daí podem conter apenas entidades inteiras ou outras composições de entidades (também inteiras).

Paralelismo

A granularidade do paralelismo presente num determinado grafo de protocolo é uma escolha de projeto que terá impacto no desempenho do protocolo como um todo. A associação entre um grafo de protocolo a um determinado ambiente de processamento virtual específico é dita uma *arquitetura de processo* [17].

De uma forma geral, quando paralelismo é introduzido com base nas tarefas do protocolo, tem-se uma arquitetura baseada em tarefas. Dependendo da granularidade do paralelismo das tarefas, a arquitetura recebe diferentes denominações. Quando os menores componentes de protocolo mapeados em clusters são sempre entidades inteiras (uma ou mais) das camadas de protocolo, tem-se então uma *arquitetura baseada em tarefas com paralelismo de camadas*. Quando os menores componentes de protocolo mapeados em cluster são tarefas internas a entidades das camadas de protocolo, tem-se uma *arquitetura baseada em tarefas com paralelismo de tarefas*. A Figura 5 exemplifica os dois tipos de paralelismo. Retângulos tracejados representam clusters e bolhas representam componentes de protocolo.

Modelo Específico do Nível de Cluster

Sendo um cluster associado a uma única thread, não há paralelismo interno em clusters. Assim, todas as operações são síncronas e o núcleo do mediador, em geral, materializa-se no próprio ambiente de execução de uma linguagem de programação orientada a objetos, que fornece mecanismos para o disparo de métodos ao estilo de chamadas de procedimentos.

Eventualmente, quando deseja-se permitir a utilização de mais de uma linguagem de programação para a implementação de objetos dentro de um mesmo cluster, objetos como stubs, skeletons e OA (vide Seção 2, na parte referente ao CORBA) devem ser incorpo-

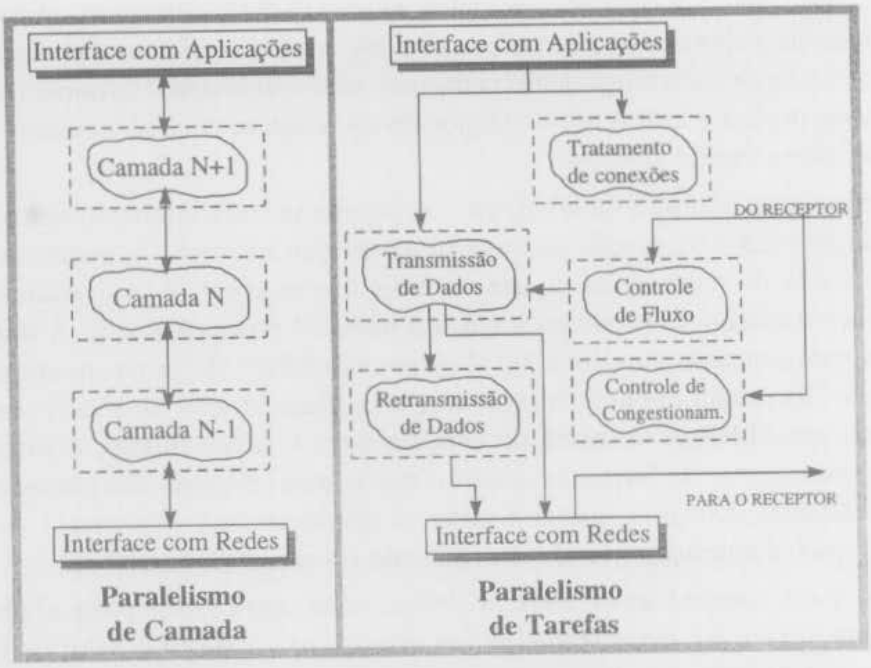

Figura 5: Arquiteturas Baseadas em Tarefas [17].

radas ao mediador para fornecer os ajustes necessários à correta troca de mensagens, identificação de objetos, etc.

Pattern Específico do Nível de Cluster

O pattern do nível de cluster define que, associado a um cluster, existe sempre um único componente de protocolo composto mais externo, denominado *objeto cluster*. A implementação de componentes compostos no nível de cluster é baseada na existência de uma estrutura em grafo, contida no componente composto, que representa o grafo de protocolo interno do componente, em que os nós são referências a componentes. Nessa estrutura, nós apresentam uma sequência, definida no momento da criação ou configuração do grafo de protocolos, que permite o acesso a cada componente através de operações do tipo `get_firstComponent()` e `get_nextComponent()`.

A implementação da operação `run()` do componente composto é formada por um laço de repetição que, sucessivamente, recupera componente a componente até o último, segundo a ordem sequencial do grafo, e solicita, a cada um, a execução da operação `run()`. No caso específico do objeto cluster, a mesma estrutura é utilizada mas o laço de repetição pode ser definido de forma a adotar uma das seguintes estratégias: (i) o laço de repetição é infinito, sendo que, sempre após o último componente do grafo, retorna-se ao primeiro; (ii) o laço de repetição dura enquanto existirem mensagens em qualquer buffer ou componente interno, sendo que, no período de duração, sempre após o último componente, retorna-se ao primeiro. No primeiro caso, denomina-se o objeto cluster de *componente ativo* e no segundo de *componente passivo*.

Projetistas podem optar por utilizar componentes ativos ou passivos da maneira que acharem mais conveniente. Componentes ativos, por exemplo, podem ser adequados quando utilizados para implementar protocolos que atendam à tráfego contínuo. Quando escalonados a taxas constantes, os componentes ativos atenderão a tráfegos dessa espécie sem a necessidade de repetidas solicitações. Na presença de tráfego variável, mesmo que pequenas interrupções no fluxo ocorram, componentes ativos não necessitam ser reativados.

Componentes passivos, por sua vez, processam a carga a que são submetidos e param. No caso de tráfego contínuo, é provável que eles se comportem de maneira análoga aos ativos, já que sempre estarão submetidos a alguma carga. Já no caso de tráfego em rajadas, componentes passivos podem ser utilizados de forma a serem ativados apenas quando necessário. Nesse caso, caberá a algum outro elemento sinalizar, avisando sempre que a operação `run()` tiver de ser solicitada a esse componente.

Já componentes ativos, quando utilizados para atender a tráfego em rajadas podem ocasionar desperdício de CPU. Por outro lado, associando-se uma prioridade de escalonamento baixa a esse componente, o desperdício deixa de existir. A possibilidade de utilizar uma prioridade baixa será dependente da qualidade do serviço desejado, que pode limitar, por exemplo, o atraso máximo das mensagens atendidas e, por conseguinte, o escalonamento.

Das considerações acima, pode-se perceber que a escolha do tipo de componente (ativo ou passivo) será o resultado da conjunção das características do tráfego e da qualidade desejada e, nesse nível, implicará na definição de determinadas políticas de escalonamento e estabelecimento de prioridades. No nível de cluster, cada vez que se define requisitos como os exemplificados acima, os pipes entre os componentes internos do objeto cluster passam a ser denominados de MediaPipes. Os controles de escalonamento não são parte do cluster, devendo ser efetuados pelo nível de cápsula ou pelo nível de nó.[1]

Modelo Específico do Nível de Cápsula

Cápsulas são ambientes de processamento virtual que exibem paralelismo interno. A interação entre objetos no nível do cluster é desconhecida pelo mediador do nível da cápsula. Objetos cluster, nesse nível, são definidos, de forma mais genérica, como as unidades de concorrência da cápsula. Pode-se, então, definir uma cápsula como um conjunto de clusters e um mediador através do qual objetos cluster (os únicos conhecidos nesse nível) podem interagir (vide Figura 6). Além disso, utiliza-se indistintamente os termos "objeto no nível de cápsula" e "objeto cluster".

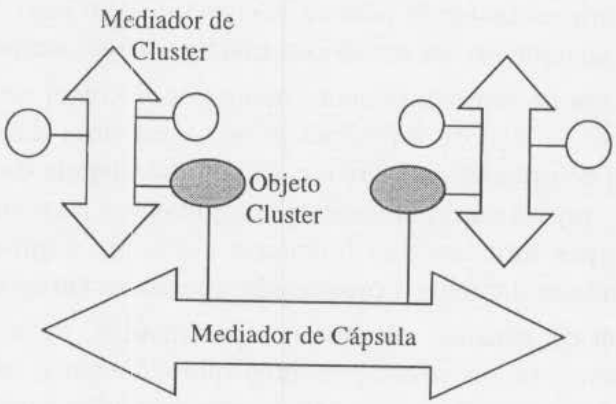

Figura 6: Cápsula como Conjunto de Clusters.

Um objeto no nível de cápsula pode atender, concorrentemente, a diversas solicitações à mesma ou a diferentes operações, tarefas e instruções. Cada objeto, nesse nível, oferece uma operação `start()`, que cria uma nova thread e executa a operação `run()`. Assim,

[1] O nível de nó corresponde ao ambiente confinado a uma máquina. Um nó gerência todas as cápsulas de uma máquina, tendo funções de escalonamento global de processos e threads. O nó também será utilizado como base para a configuração, contendo o conjunto de camadas presente na arquitetura, e operações que permitem sua manipulação. O nível de nó não será detalhado no presente trabalho.

todo o objeto que deseje oferecer concorrência deverá oferecer ambas as operações em sua interface. A solicitação de mais de um `start()` ao mesmo objeto cria diferentes threads mas não diferentes objetos, i.e., todas as threads compartilham o mesmo estado, identidade e ciclo de vida daquele objeto.

Pattern Específico do Nível de Cápsula

O pattern do nível de cápsula é, em vários aspectos, semelhante ao de cluster. Componentes compostos também são permitidos. Define-se que existe sempre um único objeto composto mais externo, denominado *objeto cápsula*, e que qualquer composição é implementada com o auxílio de uma estrutura interna que representa o seu grafo de protocolo. Novamente, nós desse grafo contém referências a componentes e estão organizados em uma seqüência definida no momento da criação ou configuração do grafo de protocolos. A operação `run()` de cada componente composto é formada por uma laço de repetição que, sucessivamente, recupera componente a componente até o último, segundo a ordem sequencial do grafo, e solicita, a cada um, a execução da operação `start()`.

Do mecanismo descrito acima, nota-se que cada operação `run()` é executada em uma thread diferente, inclusive a operação `run()` de cada componente composto do nível de cápsula. Assim, pode-se pensar num componente composto do nível de cápsula também como sendo um objeto cluster separado de seus componentes.

Se todos os componentes em um determinado componente composto forem simples e, ao mesmo tempo, ativos, o componente composto é dito *componente ativo composto* e, após o laço de repetição processar o último nó do grafo, a operação `run()` desse componente e a respectiva thread terminam. Se todos os componentes simples ou compostos em um determinado componente composto forem componentes ativos, também tem-se um componente ativo composto, e o laço se comporta da mesma forma.

Se todos os componentes em um determinado componente composto forem simples e ao mesmo tempo passivos, o laço de repetição pode seguir uma das seguintes estratégias: (i) o laço é repetido enquanto existirem mensagens nos buffers internos do componente composto, sendo esse um *componente passivo composto;* (ii) o laço de repetição é infinito, caso em que se trata, novamente, de outro componente ativo composto.

Caso os componentes de um componente composto do nível de cápsula sejam alguns passivos e outros ativos, o laço de repetição pode seguir uma das seguintes estratégias: (i) funcionar como um componente passivo, caso em que, depois de executado uma vez, o laço passa a se repetir processando apenas os componentes passivos, enquanto existirem mensagens nos seus pipes internos; (ii) funcionar como um componente ativo, caso em que o laço se repete indefinidamente processando apenas os componentes passivos.

Pipes são, no nível de cápsula, objetos compartilhados, cujas operações podem ser solicitadas concorrentemente pelos componentes que colocam e retiram mensagens. As usuais técnicas de sincronização devem, portanto, ser aplicadas para evitar inconsistências.

Uma possível associação entre um grafo de protocolo e um ambiente de processamento virtual (definindo uma arquitetura de processo) é obtida da seguinte forma: (i) associando-se entidades a objetos do nível de cápsula (compostos ou não) e (ii) associando-se entidades (de diferentes camadas), que estejam servindo a uma mesma conexão, a um objeto cápsula. Dessa forma, mensagens circulam por uma conexão em uma máquina sem troca de contexto no nível de processo. A concorrência entre processos se dá com base em mensagens de conexões diferentes, sendo essa uma arquitetura de processo denominada *arquitetura de processo baseada em mensagem com paralelismo por conexão*. Ao observar o paralelismo interno das cápsulas, nota-se a presença da arquitetura de processo baseada em tarefa, mencionada no modelo específico de cluster, na qual a troca de contexto

acontece apenas no nível de threads.

Novamente, no nível de cápsula, requisitos de tráfego e QoS são tratados a partir de políticas de escalonamento e atribuição de prioridades. Dadas as características de escalonamento dos clusters, cada thread deverá ser escalonada de acordo com alguma política determinada, que pode variar desde um escalonamento global, no qual as decisões de escalonamento são tomadas ao nível do nó, até um puramente hierárquico, no qual o nó toma decisões baseado nas cápsulas, que por sua vez, tomam decisões de escalonamento de seus clusters. No caso de escalonamento hierárquico, é função do mediador, através de um Object Adapter (OA) análogo ao definido pelo CORBA (vide Seção 2) proceder com as operações de escalonamento. Também é função do mediador efetuar o controle da viabilidade do escalonamento do sistema, aceitando ou não novos MediaPipes de acordo com as características requisitadas. Várias estratégias podem ser adotadas para esses controle como as descritas em [11, 3, 4, 12].

3.2.2 Camada

Um ambiente de processamento virtual é dito uma *camada*, quando objetos estão relacionados a uma mesma camada ou sub-camada de protocolo. Em camadas, a comunicação é sempre horizontal, sendo que, elas podem ou não ocorrer numa mesma máquina.

Modelo do Nível de Camada

Nas regras gerais de estruturação em camadas definidas pelo OSI-RM, a comunicação horizontal entre entidades pares é definida através da especificação dos serviços que uma camada (fornecedora) oferece a entidades do nível superior (solicitante e acolhedor) [23]. Essa especificação é baseada nas primitivas trocadas entre (i) solicitante e fornecedor e (ii) acolhedor e fornecedor. A especificação de um serviço, como um todo, contém as primitivas de (i) e (ii), e seu relacionamento ao longo do tempo [7].

No presente modelo, define-se dois níveis totalmente distintos, um relativo somente à interação horizontal, e outro relativo à comunicação vertical. A definição do nível de camada será tal que (i) a comunicação horizontal oferece uma abstração totalmente independente de primitivas trocadas verticalmente e (ii) a comunicação vertical mantém a estrutura de primitivas definidas pelos protocolos, permitindo a interoperabilidade com implementações já existentes.

Pattern do Nível de Camada

Uma camada N é, conceitualmente, representada por uma composição que contém todos os objetos entidade de nível N distribuídos pelo sistema e os pipes entre eles, em qualquer instante. De forma concreta, uma camada N é realizada de maneira que, em cada subsistema, tem-se um componente composto formado pelos objetos entidade de nível N presentes naquele subsistema a qualquer instante de tempo. Esse componente é denominado *(N)-camada*. Um objeto (N)-camada oferece serviços a entidades de nível N+1, as quais passa-se a denominar de (N+1)-entidades.

Cada instância de execução dos serviços de uma (N)-camada é responsabilidade de uma (N)-entidade contida nessa (N)-camada. Os serviços oferecidos são solicitados através da passagem de unidades de informação (comunicação vertical) entre essas duas entidades. Em (N)-camadas, portas são sempre as próprias (N)-entidades.

Para que uma (N+1)-entidade possa trocar unidades de informação com uma (N)-entidade, contida na (N)-camada, é necessária a configuração de dois pipes, um para

transmissão e outro para recebimento.[2] Tais pipes estão conectados, de um lado, a uma entrada ou saída da (N+1)-entidade, e de outro, a uma entrada ou saída da (N)-camada. Cada par de pipes desse gênero é denominado um *ponto de acesso ao serviço* da camada N ou *(N)-SAP* — N Layer Service Access Point. Note que, em sendo pipes, SAPs podem ser associados a especificações de tráfego e QoS, que podem ser diferentes para cada pipe do SAP (uma especificação para cada sentido). Nesse caso, SAPs passam a ser formados por MediaPipes.

No nível de camada, o modelo de comunicação horizontal segue o mesmo pattern adotado até agora: componentes de protocolo (nesse caso, objetos entidade) colocam ou retiram mensagens em pipes. Pipes, porém, são realizados de forma mais complexa, como será apresentado mais adiante. A Figura 7 ilustra esses conceitos, exemplificados numa arquitetura genérica.

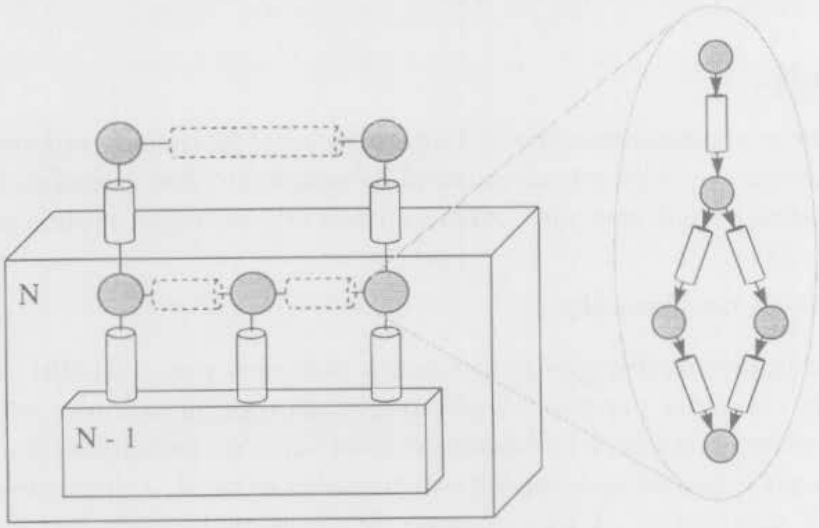

Figura 7: Arquitetura Genérica de Comunicação.

Para que a comunicação horizontal na camada N+1 seja possível, o estabelecimento de um ou mais pipes entre entidades pares se faz necessário (pipes horizontais e tracejados da Figura 7).[3] Como resultado, (N+1)-entidades recebem referências aos pipes entre elas. Embora alheias ao fato, o que as entidades recebem são, na verdade, referências a objetos criados no contexto da (N+1)-camada, denominados *PipeStubs*, que encapsulam uma referência a um (N)-SAP, e um identificador único para o pipe horizontal. A partir daí, as (N+1)-entidades colocam e retiram mensagens dos pipes horizontais como se eles fossem buffers, utilizando a referência ao pipe, que, para as elas, é o pipe horizontal. Quando isso acontece, na realidade, os PipeStubs estarão sendo acionados, os quais se encarregarão de colocar ou retirar as mensagens nos respectivos (N)-SAPs.

O estabelecimento de um MediaPipe horizontal na camada N é baseado no estabelecimento de uma conexão[4] (com as características de tráfego e QoS desejadas) através da camada N-1, seguida da solicitação do estabelecimento de MediaPipe propriamente dito sobre essa conexão. O pedido de estabelecimento de conexão à camada N-1 provoca pedidos de estabelecimento de conexões à camada N-2, e assim sucessivamente, da forma tradicional.

[2] Caso deseje-se somente comunicação unidirecional, apenas um pipe é suficiente.

[3] As formas de estabelecimento podem ser várias, desde configuração até algum mecanismo de sinalização. Para comunicação bidirecional, no mínimo dois pipes são necessários.

[4] O estabelecimento de conexões são os procedimentos usuais definidos pelos protocolos de sinalização e podem ser modelados utilizando as mesmas idéias apresentadas até aqui, formando inclusive pilhas de protocolos próprias.

O pedido de estabelecimento do MediaPipe sobre uma conexão provoca (i) a solicitação de estabelecimento de um MediaPipe vertical (um SAP) para a camada N-1 e (ii) subsequentes pedidos de estabelecimento de MediaPipes da camada N-1, sobre de conexões da camada N-2, e assim sucessivamente. Dessa forma, observa-se que MediaPipes horizontais complementam a idéia de conexões com QoS, tornando-as verdadeiramente fim a fim, já que, verticalmente, em todas as interfaces, tem-se a garantia de controle da comunicação, garantida pelos mecanismos de escalonamento, e horizontalmente, mantida pela própria implementação dos protocolos que operam sobre conexões de cada nível.

Acoplamento Vertical e Horizontal

É importante garantir que implementações que sigam os modelos e patterns apresentados possam, eventualmente se inserir em ambientes existentes. Para tanto, distigüem-se duas modalidades em que essa inserção pode ocorrer: com acoplamento vertical e com acoplamento horizontal. Essas modalidades não são exclusivas, ou seja, um ambiente pode apresentar ambas simultaneamente.

A inserção de uma implementação com acoplamento vertical está presente em praticamente todos os sistemas. Trata-se de colocar uma implementação de uma ou mais camadas de protocolo sobre ou sob uma camada já previamente implementada no ambiente. Na inserção de uma implementação com acoplamento horizontal, deseja-se inserir implementações de camadas de protocolos cuja implementação em outros subsistemas na mesma rede já existe. O modelo de referência proposto é suficientemente flexível para permitir toda a interoperabilidade desejada, em ambas as modalidades e sua combinação.

Para o acoplamento vertical, a implementação das portas das entidades da camada mais inferior da pilha a ser inserida, que ficam na interface com a camada inferior já implementada, ao invés operar leituras ou escritas sobre pipes, encapsulam a API dessa camada. Analogamente, portas das entidades da camada mais alta a ser inserida, que ficam na interface com a camada superior já implementada, ao invés operar leituras ou escritas sobre pipes, oferecem a API da interface que essa camada deseja.

Para o acoplamento horizontal, a implementação das entidades deve ser feita de forma a obedecer a especificação da máquina de estados do protocolo desejado, e o tipo relativo às mensagens dos pipes horizontais devem corresponder ao formato exato das mensagens definidas por esse protocolo.

Como um exemplo simples, ilustrado na Figura 8, considere a implementação de um cliente FTP com acoplamento vertical sobre uma implementação TCP/IP existente e acoplamento horizontal com um servidor de outra máquina.

3.2.3 Programação de Aplicações e Serviços

No nível de aplicação, tem-se um mediador genérico que deverá permitir interações entre objetos quaisquer, independente de nível de concorrência ou distribuição. O mediador, nesse caso, corresponde a um ORB, segundo definição do CORBA, estendido para o tratamento de tipos de dados contínuos e definição de QoS [2].

Os serviços do mediador para a construção de objetos com atributos de tipos contínuos devem permitir a especificação das características específicas desses atributos, que irão determinar a abstração de programação utilizada em sua manipulação. Num ambiente baseado no CORBA tradicional, a especificação de um atributo de um objeto corresponde a definição de uma variável de estado encapsulada por duas operações de acesso (tipicamente, `set` e `get`). Tais operações, aplicadas sobre objetos com tipos convencionais (não contínuos), são, em geral, tratadas por operações síncronas que lidam sempre um bloco

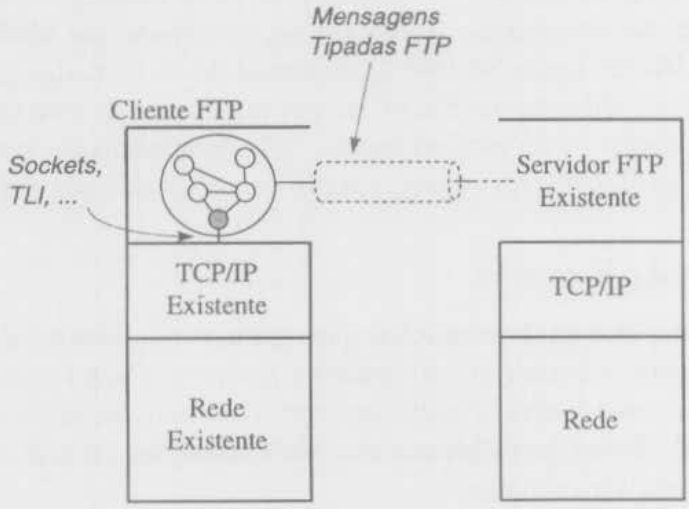

Figura 8: Acoplamento Vertical e Horizontal da Implementação de um Cliente FTP.

de informações de um tamanho determinado. Já para o tratamento de tipos contínuos, deseja-se suporte de operações assíncronas, pois a duração e o tamanho da informação são indeterminados.

O tratamento da informação a partir das operações de acesso requer um tratamento diferenciado, dependendo do desejo do programador. Considere uma operação para a recuperação de um atributo que representa um vídeo, por exemplo. Dependendo da abstração desejada, o cliente terá duas opções: (i) efetuar sucessivas chamadas `get()`, a intervalos controlados, para recuperar cada pixel ou bloco da imagem, ou (ii) efetuar uma única chamada para disparar um processo de transferência contínuo, em que ele não mais necessita interferir. A opção escolhida poderá depender de diversos fatores, incluindo o papel do objeto cliente na aplicação e a própria abstração que o programador tem do recurso utilizado. Além disso, a escolha terá influência direta na configuração de componentes ativos ou passivos nas camadas de protocolo que oferecerão suporte à comunicação.

Em relação à implementação de tipos contínuos e o controle da QoS na sua utilização, o modelo prevê extensões ao modelo do CORBA que permitem a especificação de características de continuidade em atributos de objetos e parâmetros de operações de forma que MediaPipes possam ser criados para a transferência das mensagens associadas [2]. Além disso, alguns mecanismos para o controle da carga de processamento que os objetos impõem aos ambientes são definidos.

Um objeto no nível de aplicação pode ser atendido por uma ou mais threads. A cada instrução, operação ou tarefa oferecida por um objeto associa-se uma fila de entrada. Uma mesma fila pode ser utilizada por mais de uma operação, instrução ou tarefa, mas cada operação, instrução ou tarefa só pode utilizar uma fila.

O número total máximo de threads alocadas a um objeto pode ser limitado, assim como o número de threads alocado para atendimento a cada fila. O tamanho de cada fila de entrada também pode ser especificado. Cada solicitação é colocada na respectiva fila e atendida tão logo uma thread esteja disponível, respeitando-se o número máximo total de threads e o número máximo de threads para uma mesma fila. A política de enfileiramento e descarte (nos casos em que se receba um número de solicitações maior do que o tamanho de uma fila) pode ser definido para cada fila.

4 Conclusão

Apresentou-se, no presente trabalho, um modelo de referência para a definição de arquiteturas distribuídas. Baseando-se numa visão abstrata de um modelo de orientação por objetos, modelos específicos para a interação entre objetos, com diferentes graus de distribuição e concorrência, puderam ser definidos.

Os diferentes níveis do modelo de OO especificados na visão de engenharia foram utilizados em patterns específicos para a construção de protocolos, arquiteturas ou aplicações, conforme o nível. A definição de MediaPipes e mediadores, em cada nível, garante a construção de ambientes onde as características de tráfego e QoS podem, verdadeiramente, ser mantidas fim a fim, pois tem-se a garantia de controle da comunicação vertical, assegurada pelos mecanismos de escalonamento, e horizontal, mantida pela própria implementação dos protocolos de cada nível.

A implementação dos MediaPipes sobre a idéia de conexões comuns também garante a característica de interoperabilidade com implementações de protocolo já existentes. Além disso, as características dos patterns asseguram que implementações podem ser realizadas de forma a garantir a interoperabilidade tanto horizontal, quanto vertical, com outras implementações .

Os vários patterns de estruturação e composição de componentes de protocolos permitem a sua configuração tanto estática quanto dinamicamente (em tempo de execução) embora os detalhes dessa capacidade não tenham sido detalhados neste artigo. Configuração e ciclo de vida de serviços são aspectos intimamente relacionados e serão tratados em trabalhos futuros.

O mesmo modelo de objetos abstrato foi também utilizado para descrever algumas características da construção de aplicações que podem tratar tipos contínuos. Algumas consequências em termos da necessária configuração do sistema de comunicação puderam ser apontadas.

Alguns aspectos que ainda merecem investigação cuidadosa incluem a multiplexação de conexões, a definição de planos de comunicação, o tratamento de serviços multicast, a definição patterns para o nível de aplicação e a definição de mecanismos de escalonamento específicos.

Referências

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns.* Wiley, 1995.

[2] S. Colcher, L. F. G. Soares, M. A. Casanova, and G. L. Souza. Modelo de objetos baseado no CORBA para sistemas hipermídia abertos com garantias de sincronização. In M. Oliveira, editor, *XIV Simpósio Brasileiro de Redes de Computadores (SBRC)*, pages 18–37, Fortaleza, CE, Maio 1996.

[3] B. Ford and S. Sai. CPU Inheritance Scheduling. In *Operating Systems Design and Implementation (OSDI'96)*, Seattle, October 1996.

[4] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU Scheduler for Multimedia Operating Systems. In *Operating Systems Design and Implementation (OSDI'96)*, pages 107–122, Seattle, October 1996.

[5] H. Huni, R. Johnson, and R. Engel. A framework for network protocol software. In *OOPSLA'95.* ACM SIGPLAN Notices, 1995.

[6] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17:64–76, January 1991.

[7] ISO/IEC 10731 / ITU-T X.210. *Information Technology - Open Systems Interconnection - Basic Reference Model: Conventions for the Definition of OSI Services*, 1993.

[8] ISO/IEC 7498 / ITU-T X.200. *Information Processing Systems - Open Systems Interconnection - Basic Reference Model*, 1984.

[9] ISO/IEC JTC1/SC21/WG7. *ISO/IEC DIS 10746-1/ITU-T X.901 — Reference Model of Open Distributed Processing, Part 1: Overview*, may 1995. output from the editing meeting in Helsink (Finland).

[10] A. A. Lazar, K. S. Lim, and F. Marconcini. Binding model: Motivation and description. http://www.ctr.columbia.edu/comet/xbind/xbind.html, November 1995.

[11] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Jornal of the ACM*, 20(1):46–61, January 1973.

[12] A. Mauthe and G. Coulson. Scheduling and Admission Testing for Jitter Constrained Periodic Threads: Discussion and Proof. Technical Report MPG-96-12, Distributed Multimedia Research Group, Department of Computing, Lancaster University, Lancaster LA1 4YR, UK, 1996.

[13] R. Meunier. The pipes and filters architecture. In J. Coplien and D. Schmidt, editors, *Pattern Languages of Programming Design*, pages 427–440. Addison Wesley, 1995.

[14] D. Mosberg. Message library design notes. x-Kernel Internal Report, Jan. 1996.

[15] P. Nikander, J. Parssinen, B. Sahlin, and K. Hoglund. A java based framework for criptographic protocols. Technical report, Telecommunications Software and Multimedia, Helsinki University of Technology, May 1997. Draft Research Report.

[16] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification*, July 1995. Revision 2.0 (updated in July, 1996).

[17] D. C. Schmidt. The ADAPTIVE communication environment: An Object-Oriented network programming toolkit for developing communication software. In *Proceedings of the 12th Annual Sun Users Group Conference*, pages 214–225, San Jose, CA, December 1993. SUG.

[18] D. C. Schmidt. Applying design patterns and frameworks to develop Object-Oriented communication software. In P. Salus, editor, *Handbook of Programming Languages*, volume I. MacMillan Computer Publishing, 1997.

[19] D. C. Schmidt and C. Cleeland. Applying patterns to develop and maintain extensible ORB middleware. *Communications of ACM*, 40(12), December 1997. This article will apear in the CACM Special Issue on Software Maintenance.

[20] D. C. Schmidt, A. Gokhale, T. H. Harrison, D. Levine, and C. Cleeland. TAO: a High-Performance ORB endsytem architecture for Real-time CORBA. Document submitted as an RFI response to the OMG Special Interest Group on Real-time CORBA.

[21] D. C. Schmidt, A. Gokhale, T. H. Harrison, and G. Parulkar. A High-Performance endsystem architecture for Real-Time CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[22] L. F. G. Soares, M. A. Casanova, and S. Colcher. An architecture for hypermedia systems using MHEG standard objects interchange. *Information Services & Use*, 13(2):131–139, 1993. Special Issue: Hypermedia and Hypertext Standards.

[23] L. F. G. Soares, G. Lemos, and S. Colcher. *Redes de Computadores: das LANs, MANs e WANs às Redes ATM*. Ed. Campus, 1995. Segunda Edição.

[24] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. Keynote session of Multimedia Computing and Networking Conference, San Jose, CA, January 1996.