PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Marcelo Ferreira Moreno

Um Framework para Provisão de QoS em Sistemas Operacionais

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática da PUC-Rio.

Orientador: Prof. Luiz Fernando Gomes Soares

Rio de Janeiro
agosto de 2002

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Marcelo Ferreira Moreno

Um Framework para Provisão de QoS em Sistemas Operacionais

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Luiz Fernando Gomes Soares
Orientador
Departamento de Informática – PUC-Rio

Prof. Sérgio Colcher
Departamento de Informática – PUC-Rio

Prof. Markus Endler
Departamento de Informática – PUC-Rio

Prof. Ney Dumont
Coordenador Setorial do Centro
Técnico e Científico – PUC-Rio

Rio de Janeiro, 16 de agosto de 2002

Marcelo Ferreira Moreno

Graduado em Informática pela Universidade Federal de Viçosa, onde desenvolveu projetos na área de inteligência artificial, manipulando redes neurais. Desde dezembro de 2001, é colaborador de um projeto internacional de código livre, cujo objetivo é a criação de uma interface para a configuração dos mecanismos de qualidade de serviço presentes no sistema operacional Linux. Atualmente, integra o grupo de pesquisadores do Laboratório TeleMídia da PUC-Rio, trabalhando nas áreas de sistemas operacionais, redes de computadores e aplicações distribuídas multimídia.

Ficha Catalográfica

Moreno, Marcelo Ferreira

Um framework para provisão de QoS em sistemas operacionais / Marcelo Ferreira Moreno; orientador: Luiz Fernando Gomes Soares. – Rio de Janeiro : PUC, Departamento de Informática, 2002.

[12], 105 f. : il. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Qualidade de Serviço. 3. Sistemas operacionais, 4. Adaptabilidade. 5. Frameworks. 6. Controle de admissão. 7. Escalonamento de recursos. I. Soares, Luiz Fernando Gomes. II. Pontifícia Universidade Católica do Rio de Janeiro. III. Título.

CDD: 004

Este trabalho é dedicado

A Lorenza, pela cumplicidade, respeito, carinho e amor sempre presentes em nossas vidas, unidas pela graça de Deus.

Aos meus pais, Wanderley e Jasmina, pela compreensão e incentivo em todos os momentos decisivos de minha vida, traçada por eles sobre o caminho do bem.

Aos amigos Antônio Moisés e Maria do Carmo, por me acolherem como um filho e por confiarem a mim o bem mais precioso de suas vidas.

A Deus, pela luz que me guia.

Agradecimentos

Agradeço ao meu orientador, Professor Luiz Fernando Gomes Soares, a oportunidade, as valiosas contribuições e o emprego de sua experiência, fundamentais para a conclusão deste trabalho. Agradeço, também, a sua amizade, incentivo e ajuda pessoal, sempre presentes nos momentos difíceis dessa caminhada.

Obrigado aos colegas do Laboratório TeleMídia, pelo ambiente agradável de trabalho e pela disposição em ajudar no surgimento de dúvidas e problemas. Em especial, quero agradecer aos amigos Alésio Pfeifer e Luciana Lima, pelo companheirismo mútuo e pelos imprescindíveis momentos de descontração. A Antônio Tadeu Gomes agradeço pela sua disponibilidade para esclarecimentos sobre seus trabalhos anteriores e pelas contribuições que ajudaram a definir os rumos da minha pesquisa.

Aos meus padrinhos Rômulo e Elizete Siqueira, obrigado pela acolhida no meu primeiro mês de curso, pois, quando várias eram as minhas incertezas, foram o meu porto seguro.

Agradeço também aos professores e funcionários da PUC-Rio que, de alguma forma, colaboraram no decorrer deste trabalho, e que fazem dessa instituição um centro de excelência reconhecido internacionalmente. Finalmente, agradeço à CAPES, pelo financiamento do meu trabalho a partir de seu programa de bolsas de mestrado, e à PUC-Rio, pela concessão da bolsa de isenção de taxas escolares.

Resumo

Moreno, Marcelo Ferreira; Soares, Luiz Fernando Gomes. **Um Framework para Provisão de QoS em Sistemas Operacionais**. Rio de Janeiro, 2002. 105p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A demanda progressiva por aplicações multimídia distribuídas, caracterizadas por fortes exigências sobre os recursos computacionais, torna evidente a necessidade de provisão de qualidade de serviço (QoS) em cada um dos subsistemas envolvidos, como redes de comunicação e sistemas operacionais. Ao mesmo tempo, tais subsistemas devem ser flexíveis para que possam oferecer novos serviços a aplicações futuras, ou seja, devem ser adaptáveis em tempo de execução. Especificamente, sistemas operacionais de uso geral provêm pouco ou nenhum suporte a QoS e à adaptabilidade dos serviços, impulsionando vários estudos isolados nessas áreas. Observando-se algumas tecnologias implementadas em sistemas operacionais específicos, nota-se que os mecanismos de provisão possuem certas semelhanças funcionais. Assim, este trabalho propõe uma arquitetura adaptável para a provisão de QoS nos subsistemas de rede e de escalonamento de processos de sistemas operacionais, independente de implementação, através da descrição de frameworks genéricos. É demonstrado, também, como os pontos de flexibilização desses frameworks podem ser especializados para a implementação de alguns modelos de QoS. Por último, é proposto um cenário de uso da arquitetura, no qual um sistema operacional de uso geral ligeiramente modificado é utilizado como infra-estrutura para a instanciação dos frameworks de QoS.

Palavras-chave

Qualidade de Serviço, Sistemas Operacionais, Adaptabilidade, Frameworks, Controle de Admissão, Escalonamento de Recursos

Abstract

Moreno, Marcelo Ferreira; Soares, Luiz Fernando Gomes. **A Framework for QoS Provisioning in Operating Systems**. Rio de Janeiro, 2002. 105p. MSc. Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro

The progressive demand for distributed multimedia applications, which are characterized by strong requirements over computational resources, makes evident the need for quality of service (QoS) provisioning in each one of the involved subsystems (e.g. communication networks and operating systems). At the same time, these subsystems must be flexible enough that they can offer new services to future applications, or in other words, they must be adaptable at runtime. Specifically, general-purpose operating systems provide few or no QoS/service adaptability support, what have motivated many isolated studies about these topics. Observing some implemented technologies on specific operating systems, it is noted that the provisioning mechanisms have certain functional similarities. In this way, this work proposes an adaptable architecture for QoS provisioning on networking and process scheduling subsystems of operating systems, through the description of generic frameworks. It is demonstrated how the framework *hot-spots* can be specialized in order to implement some QoS models. Finally, it is proposed a scenario of use of the architecture, where a bit modified general-purpose operating system is used as infrastructure for an instantiation of the QoS frameworks.

Keywords

Quality of Service, Operating Systems, Adaptability, Frameworks, Admission Control, Resource Scheduling

Sumário

Lista de Figuras

Lista de Tabelas

1
Introdução

O termo Qualidade de Serviço (QoS) tem sido muito referenciado nos últimos anos, devido, principalmente, à crescente demanda por aplicações distribuídas de alto desempenho, que exigem certos requisitos do sistema, como retardo máximo, variação estatística máxima do retardo, taxa mínima de transmissão, entre outros. Oferecer um serviço com QoS significa permitir que tais parâmetros sejam garantidos às aplicações, de acordo com os valores por elas fornecidos, durante seu período de execução.

A provisão de qualidade de serviço em sistemas caracterizados por fortes exigências de processamento e comunicação impõe a necessidade de atendimento aos requisitos de maneira fim-a-fim, ou seja, cada subsistema componente do serviço deve ter seus recursos gerenciados no intuito de garantir aos usuários o nível de qualidade por cada um deles solicitado (Gomes, 1999). Numa aplicação multimídia distribuída, por exemplo, não só o sistema de comunicação em rede deve oferecer os mecanismos para a provisão de QoS, mas também o sistema operacional das estações envolvidas e, recursivamente, os seus subsistemas relevantes. Nas estações finais, recursos controlados pelo sistema operacional, como CPU, memória e buffers de comunicação, devem ser gerenciados de forma a assegurar que a coexistência de várias aplicações não viole as necessidades individuais de QoS de cada uma delas.

Para provisão de QoS, é interessante que os sistemas envolvidos, em particular os sistemas operacionais, sejam flexíveis, para que novos serviços possam ser configurados visando sua utilização por futuras aplicações. A especificação de um novo serviço pode envolver a escolha de algoritmos de escalonamento, admissão e classificação, por exemplo, no que se refere diretamente ao uso dos recursos de processamento e comunicação. Outros parâmetros de configuração podem ser citados, como as tarefas que irão compor a pilha de protocolos de comunicação ou a descrição do estado inicial do sistema

para a provisão de QoS (conjunto de algoritmos citados e particionamento inicial dos recursos para cada tipo de aplicação). Em outras palavras, é desejável que os sistemas possam ser adaptáveis, para o oferecimento de novas modalidades de serviço, a partir da modificação de seu estado interno em tempo de operação.

O processamento por compartilhamento do tempo e a estrutura monolítica em que se baseiam os sistemas operacionais de uso geral criam, contudo, várias barreiras para adaptabilidade[1] e provisão de QoS. Os escalonadores de processos desses sistemas recorrem apenas ao expediente do uso de prioridades para privilegiar a execução de algumas aplicações em detrimento de outras. Seus subsistemas de rede são guiados por interrupções, o que pode causar anomalias de escalonamento de processos. Normalmente, as filas de transmissão de pacotes são compartilhadas, não havendo meios para a classificação ou priorização dos pacotes. Por último, são raros os mecanismos para a reserva de recursos e para a introdução de partes adaptáveis no *kernel*[2] em tempo de execução.

Visto que os sistemas operacionais de uso geral possuem pouco ou nenhum suporte para a adaptabilidade e provisão de QoS, muitas foram as pesquisas desenvolvidas nessas áreas. Alguns trabalhos integram projetos para a construção de novos sistemas, estruturados desde suas concepções para desempenharem tais funções. Outros trabalhos propõem a extensão de sistemas operacionais já existentes, sejam eles de uso geral ou não. Examinando-se vários desses estudos, percebe-se que os mecanismos propostos possuem semelhanças funcionais entre si, o que evidencia a possibilidade de serem descritos de forma genérica. O desenvolvimento de uma arquitetura genérica para provisão de QoS é de grande utilidade, uma vez que pode clarificar os conceitos envolvidos, facilitar a reutilização das funcionalidades comuns e definir uma organização interna que seja equivalente nos diferentes sistemas.

---

[1] Nota-se que o termo adaptabilidade, no contexto de qualidade de serviço, é utilizado em muitos trabalhos para descrever a capacidade de reação dos subsistemas componentes do serviço mediante variações apresentadas na carga sobre os recursos envolvidos. Esse conceito será introduzido no Capítulo 3, com a denominação de sintonização da QoS, evitando qualquer ambigüidade sobre o termo “adaptabilidade” neste trabalho.

[2] Neste trabalho, será usado o substantivo kernel da língua inglesa, para denominar o núcleo do sistema operacional, ou seja, o conjunto de rotinas que integram o controle central da estação.

1.1 Objetivos da Dissertação

O objetivo principal deste trabalho é a descrição de uma arquitetura genérica adaptável para a provisão de QoS nos subsistemas de rede e de escalonamento de processos de sistemas operacionais, baseada nos *frameworks para provisão de QoS em ambientes genéricos de processamento e comunicação,* apresentados em (Gomes, 1999).

O trabalho referenciado acima identifica as funções recorrentes de provisão de QoS nos vários ambientes envolvidos (e.g. redes de comunicação, sistemas distribuídos e sistemas operacionais) e como essas funções participam da orquestração de recursos para o fornecimento de serviços com QoS verdadeiramente fim-a-fim. Os componentes foram estruturados sob a forma de frameworks no intuito de facilitar a identificação dos pontos de flexibilização (*hot-spots*), que devem ser preenchidos para descrever a funcionalidade de um ambiente específico.

Particularmente, o presente trabalho mostra como alguns desses pontos de flexibilização podem ser completados para acomodar várias técnicas de provisão de QoS em sistemas operacionais. Por outro lado, outros pontos de flexibilização são deixados em aberto para que a arquitetura atenda aos requisitos de generalidade e de adaptabilidade a novos serviços, possibilitando a configuração de certas funções já citadas anteriormente.

Outro objetivo deste trabalho é mostrar que o sistema Linux (Rusling, 1996), apesar de caracterizado pelas desvantagens comuns aos sistemas operacionais de uso geral, pode ser ligeiramente modificado para que alguns mecanismos de provisão de QoS possam ser oferecidos. Dessa forma, a arquitetura proposta pode ser aplicada em um cenário de uso sobre este sistema de código aberto.

O enfoque do presente trabalho, definido sobre os subsistemas de comunicação e processamento de sistemas operacionais, foi motivado pela importância da provisão de QoS para aplicações multimídia distribuídas, caracterizadas pelo uso em massa dos recursos desses dois subsistemas. Nota-se,

porém, que recursos como memória principal, memória virtual (sistema de paginação), discos (memória secundária), interfaces de vídeo e de som são, também, muito importantes para esse tipo de aplicação, o que torna necessária uma orquestração também entre esses recursos. Por ser um assunto extremamente amplo, optou-se pela restrição citada acima, fundamentada no fato de que diversos trabalhos relacionados apresentaram grande aumento na eficiência do sistema como um todo, propondo soluções apenas para o escalonamento de processos e o processamento da pilha de protocolos.

Reforçando este empirismo, é observado que existe uma ordem natural para o desenvolvimento de mecanismos de provisão de QoS em sistemas operacionais. Por exemplo, menor será o ganho na implementação de um subsistema de paginação de memória com garantias de QoS se, antecipadamente, não forem tratadas as questões acerca da própria execução dos processos. O gerenciamento eficiente da preempção das aplicações e do processamento da pilha de protocolos pode, adicionalmente, levar a uma redução do número de mudanças de contexto e, conseqüentemente, do número de páginas a serem buscadas na memória virtual.

1.2 Estrutura da dissertação

Esta dissertação encontra-se estruturada como se segue. No Capítulo 2, são descritas as características dos subsistemas de rede e de escalonamento de processos que são relevantes no estudo da provisão de qualidade de serviço em ambientes multimídia distribuídos. Primeiramente, é feito um rápido estudo sobre os sistemas operacionais de uso geral, para que possam ser apontadas as particularidades que representam empecilhos para a provisão de QoS. Em seguida, são apresentados alguns trabalhos relacionados a esses pontos críticos, além de arquiteturas alternativas para a construção de sistemas operacionais com suporte a qualidade de serviço. Essas informações são importantes para que sejam identificados os pontos de flexibilização dos frameworks propostos e para que seja construída uma modelagem genérica do funcionamento do sistema, baseada em redes de filas estendidas (Soares, 1990). Tal modelagem tem o objetivo de relacionar os principais recursos de processamento e comunicação envolvidos sob

o gerenciamento de um sistema operacional e expor as dependências existentes entre eles.

O Capítulo 3 apresenta a arquitetura proposta, mostrando como os frameworks descritos em (Gomes, 1999) podem ser especializados ou mesmo estendidos para a definição de um modelo adaptável aos diversos mecanismos de provisão de QoS em sistemas operacionais. Acompanhando a descrição dos componentes de cada framework são mostrados exemplos de sua instanciação para um contexto específico.

Em seguida, o Capítulo 4 propõe um cenário real de uso da arquitetura, sobre o sistema operacional Linux. Além da instanciação dos frameworks definidos no Capítulo 3, são descritas as modificações e configurações necessárias sobre o sistema, para que sejam oferecidos alguns mecanismos para adaptabilidade e provisão de QoS.

Por fim, no Capítulo 5 são feitas as considerações finais sobre o trabalho, destacadas as contribuições da dissertação e relacionados os possíveis trabalhos futuros.

2
QoS em Sistemas Operacionais

Este Capítulo tem o objetivo de evidenciar as deficiências encontradas nos subsistemas de rede e de escalonamento de processos de sistemas operacionais de propósito geral (GPOS[3]), no que se refere ao oferecimento de serviços com QoS, e mostrar as soluções descritas em vários trabalhos relacionados. Entre esses trabalhos, foram encontrados três tipos principais de estudo: estudos isolados sobre os subsistemas citados, independentes do sistema operacional; estudos para o desenvolvimento de novos sistemas operacionais; e estudos para a extensão dos sistemas operacionais existentes, principalmente os GPOS.

Identificando os principais mecanismos de provisão de QoS descritos por esses trabalhos, podem ser determinados seus aspectos em comum e, assim, fundamentar a construção dos frameworks propostos. A terminologia utilizada para a descrição das funcionalidades em questão é a mesma definida em (Gomes, 1999).

Primeiramente, serão apresentadas as fases que compõem a provisão de QoS no contexto de sistemas operacionais, descrevendo quais as funções envolvidas em cada uma delas. Após a análise sobre os GPOS e trabalhos relacionados, será feita uma modelagem genérica do funcionamento do sistema, onde estarão representados os subsistemas de processamento e comunicação e seus recursos, que participam do oferecimento de serviços às aplicações. Será identificado um importante relacionamento entre esses subsistemas, que deve ser considerado na construção dos frameworks propostos no Capítulo 3.

---

[3] GPOS – Sigla para General Purpose Operating System – será utilizada para identificar os sistemas operacionais de propósito geral, tanto no singular quanto no plural. Os GPOS são os sistemas encontrados comercialmente com certa popularidade e que não se dedicam a tarefas e aplicações específicas.

2.1
Fases da Provisão de QoS

A provisão de QoS requer que o fornecedor do serviço, no caso o sistema operacional, ofereça certos mecanismos que serão utilizados em fases bem definidas do ciclo de vida de um serviço. Tais fases serão descritas a seguir, destacando as funções e estruturas básicas necessárias em cada uma delas.

- Iniciação do Provedor de Serviços
- Solicitação de Serviços
- Estabelecimento de Contratos de Serviço
- Manutenção de Contratos de Serviço
- Encerramento de Contratos de Serviço

2.1.1
Iniciação do Provedor de Serviços

A iniciação do provedor de serviços compreende a *identificação dos recursos disponíveis* para o fornecimento dos serviços e a *definição do estado interno inicial* do provedor. É desejável que a associação dos recursos aos serviços seja feita dinamicamente, no momento da solicitação, tanto para que os recursos possam ser alocados de forma mais eficiente, quanto para que o sistema ofereça uma maior flexibilidade aos usuários. Associações estáticas, no entanto, podem ser feitas em tempo de projeto e ativadas nesta fase de iniciação.

Em sistemas operacionais, a iniciação pode ser feita no momento de carga do sistema, quando o *kernel* conhecerá os recursos disponíveis e poderão ser determinadas as suas estruturas iniciais para alocação. Dependendo da *adaptabilidade* do sistema, o serviço poderá ser reiniciado a partir de um novo estado interno[4] inicial, sem que o sistema operacional tenha que ser recarregado. Em um nível de adaptabilidade ainda maior estariam os sistemas que podem *incluir novos serviços em tempo de operação*, agregando um novo conjunto de

[4] O estado interno de um sistema compreende a situação de particionamento de cada um dos recursos gerenciados e os conjuntos de políticas de QoS para uso pelos serviços oferecidos.

políticas de provisão de QoS ao seu estado interno anterior. Dessa forma, o sistema operacional terá a capacidade de adotar, por exemplo, novas estratégias de escalonamento e de admissão de recursos.

2.1.2 Solicitação de Serviços

Completada a fase de iniciação, o provedor se encontra pronto para receber as solicitações de serviços dos usuários. Num sistema operacional, a solicitação pode partir diretamente da aplicação ou pode ter sido repassada por um outro nível de abstração, através, por exemplo, de um protocolo de negociação de QoS. A solicitação contém os parâmetros que correspondem à *caracterização do tráfego* para os fluxos[5] gerados pelo usuário e à *especificação da QoS* necessária.

Dependendo do *nível de abstração* no qual estão definidos os parâmetros da solicitação, o sistema deve possuir funções de *mapeamento* capazes de traduzi-los para valores relacionados diretamente com a capacidade de processamento ou comunicação dos recursos. Por exemplo, no nível da aplicação, a especificação da QoS de um sistema de vídeo sob demanda poderia envolver a taxa de atualização do vídeo e o tamanho de seu quadro. No nível do sistema, esses parâmetros devem ser mapeados para a taxa de pico de transmissão de bits através de um enlace de rede e para a taxa de instruções executadas por uma CPU para tratar o vídeo.

2.1.3 Estabelecimento de Contratos de Serviço

Após a solicitação de um serviço, o provedor deve executar suas funções de *controle de admissão* para verificar a viabilidade da aceitação de um novo fluxo, de forma que a QoS solicitada seja garantida. As *estratégias de admissão* se baseiam em informações sobre as *reservas de recursos* já efetuadas ou em

[5] Neste trabalho, um fluxo pode representar uma seqüência de unidades de informação (e.g. pacotes de dados) transmitida entre processos (dispostos local ou remotamente) ou uma seqüência de instruções a ser executada pelo microprocessador de uma estação. Esta diferenciação será discutida na Seção 2.3.

medições da utilização real dos recursos para determinar se o novo serviço poderá ser aceito, dadas a especificação da QoS e a caracterização do tráfego gerado.

De acordo com a *categoria de serviço* solicitada, as estratégias de admissão podem se comportar de maneira conservadora ou agressiva. Estratégias de admissão conservadoras são aquelas que calculam a reserva dos recursos de forma menos eficiente, permitindo um número bem menor de fluxos coexistentes, mas com fortes garantias de *manutenção da QoS*. Já as estratégias agressivas avaliam as reservas de forma mais inteligente, admitindo um maior número de fluxos, mas correndo um certo risco de ocorrência de sobrecarga sobre o recurso em algum momento.

Se o controle de admissão concluir que a solicitação do serviço não pode ser atendida, o *contrato de serviço* não é estabelecido, podendo o usuário realizar uma requisição equivalente em um outro momento ou, imediatamente, especificar parâmetros mais relaxados para o serviço.

Por outro lado, se a admissão for bem sucedida, são acionados os métodos para a criação dos *recursos virtuais*[6], através da configuração dos escalonadores de recursos, dos classificadores e dos agentes de policiamento. Assim, implicitamente está estabelecido o contrato de serviço, que obriga o provedor a manter o nível de QoS solicitado ao longo do período de provisão e o usuário a submeter seu fluxo em conformidade com a caracterização informada.

As operações que compõem a fase de estabelecimento de contratos de serviços são providas por mecanismos de *negociação de QoS*. Esses mecanismos também são responsáveis pela *orquestração dos recursos*, que consiste na divisão de responsabilidade da negociação de QoS entre os subsistemas que integram o provedor de serviços ou entre os múltiplos recursos de um mesmo subsistema.

Particularmente, em sistemas operacionais ocorrem ambas situações de orquestração de recursos. Por exemplo, em aplicações multimídia distribuídas, o negociador de QoS de um sistema operacional deve delegar a negociação aos

[6] Um recurso virtual corresponde à parcela reservada do recurso para uso exclusivo do fluxo submetido pelo usuário. Este conceito será melhor discutido no Capítulo 3.

mecanismos específicos dos principais subsistemas envolvidos, no caso, os subsistemas de rede e de gerenciamento de processos. Por sua vez, o subsistema de rede pode abranger o gerenciamento de várias filas de pacotes presentes em diferentes níveis da pilha de protocolos de rede. O negociador de QoS do subsistema de rede deve, então, dividir sua responsabilidade entre negociadores que poderiam estar associados a cada um dos níveis de protocolo de rede, onde houver retenção de pacotes para a comunicação com o próximo nível da pilha.

2.1.4 Manutenção de Contratos de Serviço

Para manter os acordos estabelecidos pelos contratos de serviço, o provedor deve assegurar que os serviços sejam oferecidos conforme as especificações de QoS solicitadas e que os fluxos submetidos pelos usuários estejam conformes com relação aos perfis de tráfego fornecidos. O comportamento inadequado de ambas as partes frente ao contrato estabelecido estará sujeito a penalidades que vão de uma simples notificação ao usuário até a interrupção abrupta da provisão do serviço.

Em sistemas operacionais, os mecanismos necessários para a operação de um serviço com QoS compreendem as funções de *classificação dos fluxos* e *escalonamento de recursos*. A manutenção dos contratos envolve a *monitoração* e *sintonização de QoS*.

A classificação dos fluxos tem o objetivo de identificar a categoria de serviço a que pertence uma dada unidade de informação componente do fluxo e, conseqüentemente, o escalonador de recursos a ser utilizado. Um exemplo é o classificador de pacotes de rede, que identifica a categoria de serviço de um pacote analisando dados contidos no seu cabeçalho.

Os escalonadores de recursos gerenciam o compartilhamento do recurso entre os vários fluxos que necessitam utilizá-lo. Esse compartilhamento deve seguir as especificações de QoS de cada usuário, protegendo-os individualmente das sobrecargas que podem ser geradas por alguns deles. Para isso, é comum o

emprego de vários algoritmos de escalonamento sobre um mesmo recurso, dadas as diferentes necessidades de QoS impostas pelas aplicações.

Como forma de verificar a conformidade dos fluxos gerados pelo usuário e a existência de sobrecarga em certos recursos, o sistema utiliza a monitoração da QoS. O policiamento de fluxos é um tipo de monitoração que faz medições do tráfego gerado pelo usuário e pode tomar decisões que evitem a violação da QoS especificada. Por exemplo, o policiamento de um fluxo de pacotes de rede detecta quais desses pacotes estão além da caracterização de tráfego informada, podendo apenas marcá-los como não-conformes ou descartá-los definitivamente, evitando que sejam processados.

Já os mecanismos de monitoração do próprio sistema verificam a carga sobre os recursos e qual a real QoS oferecida a cada um dos fluxos. Situações de sobrecarga podem ocorrer devido a estratégias de reserva de recursos pouco conservadoras ou pela falha de algum componente do sistema. Caso seja realmente detectada uma violação do contrato estabelecido, o sistema pode tomar algumas decisões, como apenas alertar os usuários afetados, ou dar início a uma renegociação de QoS, ou, ainda, ativar mecanismos de sintonização de QoS.

A sintonização de QoS envolve uma nova orquestração dos recursos, quando podem ser utilizados subsistemas ou recursos alternativos (e.g. filas de interfaces de rede que podem ser uma alternativa para a comunicação desejada), apesar de serem pouco comuns em sistemas operacionais. Pode-se, também, redistribuir a carga entre os recursos, ajustando-se seus parâmetros de escalonamento.

2.1.5 Encerramento de contratos de serviços

Findado o interesse de uma das partes em utilizar/prover o serviço, ou ainda, expirado o tempo de duração especificado no contrato, dá-se início à fase de encerramento do contrato de serviço. Recebida a requisição de término, gerada pelo usuário ou automaticamente, o provedor deve liberar, em cada um dos

subsistemas envolvidos com o fornecimento do serviço, os recursos reservados e seus classificadores e policiadores.

2.2 Características relevantes de sistemas operacionais de uso geral

A principal função de um sistema operacional é o gerenciamento dos recursos computacionais de uma estação, disponibilizando-os para quaisquer aplicações que necessitem realizar a transferência de dados entre eles. Alguns exemplos desses recursos são: microprocessadores, memórias, dispositivos de armazenamento secundário, buffers de comunicação e interfaces de entrada e saída de dados.

Os GPOS são sistemas de tempo compartilhado, no qual várias aplicações disputam simultaneamente o uso dos recursos citados acima. A evolução desses sistemas é caracterizada, especialmente, pela otimização do tempo despendido no controle dessa concorrência. Por isso, os desenvolvedores vinham trabalhando, principalmente, na redução de complexidade dos algoritmos de escalonamento de recursos, na diminuição do tamanho do código do *kernel* e no uso otimizado das propriedades específicas da arquitetura de hardware, como instruções exclusivas de certos microprocessadores.

Porém, várias características dos sistemas operacionais de uso geral, incluindo a simplicidade com que é tratada a alocação de recursos, impossibilitam o oferecimento de serviços com QoS a aplicações que possuem fortes exigências de processamento e comunicação. Nesta Seção, serão discutidas as características dos GPOS que influenciam, impossibilitam, ou são ausentes, para a provisão de qualidade de serviço e suporte à adaptabilidade.

2.2.1 Arquitetura de sistemas operacionais

A maioria dos GPOS, tais como Linux e Windows, está organizada internamente em uma estrutura monolítica, ou seja, é formada por um grande núcleo que abrange todas as funções de controle do sistema. Funções como

escalonamento de processos, gerenciamento de memória, comunicação entre processos e pilha de protocolos estão todas elas implementadas no *kernel*. Historicamente, a escolha por essa arquitetura se deve à facilidade de comunicação entre os módulos internos, através de chamadas de função com passagem de parâmetros, dando grande eficiência ao sistema (Maxwell, 2000). Outra vantagem dos sistemas monolíticos é a forte associação com as funcionalidades específicas da plataforma de hardware, tirando um melhor proveito dos benefícios oferecidos por cada uma delas.

Por outro lado, a arquitetura monolítica apresenta grande dificuldade de substituição de módulos internos por implementações mais eficientes, para um dado tipo de serviço, em tempo de projeto ou em tempo de execução. A capacidade de inserção e alteração de parte do código do *kernel* durante o funcionamento de um sistema operacional integra uma das formas de suporte à *adaptabilidade* do sistema para a inclusão de novos serviços. Além disso, a portabilidade do sistema se torna uma operação extremamente complexa, já que a migração do código de um *kernel* monolítico para uma outra plataforma de hardware leva à modificação de grande parte do seu extenso código.

Os GPOS se caracterizam, ainda, pelo processamento não-preemptivo do kernel, ou seja, a execução das instruções que integram o núcleo do sistema não pode ser interrompida para que a CPU seja utilizada por um outro processo. Por exemplo, um processo de alta prioridade na fila de execução da CPU deve esperar que uma chamada de sistema, disparada anteriormente por um processo de baixa prioridade, seja executada até seu fim, para que ocorra a preempção. Essa situação é chamada *inversão de prioridade* e é agravada pelo grande número de instruções que podem compor a execução de uma chamada de sistema em um *kernel* monolítico. Na provisão de serviços com QoS, a ocorrência de inversão de prioridades deve ser reduzida ao máximo.

2.2.2 Escalonamento de processos

As aplicações em execução em uma estação são representadas no sistema operacional por estruturas chamadas processos. Depois de carregado, um processo

entra em uma fila de espera pelo uso da CPU, chamada de fila de prontos, contendo processos no estado executável (runnable). O controle de acesso dos vários processos em execução à CPU é feito pelo escalonador de processos, que arbitra os intervalos de tempo a que eles terão direito.

O escalonamento de processos em um sistema operacional está sujeito a vários tipos de aplicações, que possuem diferentes necessidades de processamento. Em (Goyal, 1996b) são descritas três categorias principais de aplicações que podem coexistir em um sistema multimídia, considerando os requisitos quanto ao tempo de uso da CPU:

- Aplicações em tempo real severo (hard real-time): São aquelas que requerem garantias determinísticas sobre o retardo característico de ambientes multitarefa. Ex.: Controle distribuído de processos industriais.
- Aplicações em tempo real suave (soft real-time): Requerem garantias estatísticas sobre os parâmetros de qualidade de serviço, tais como retardo máximo e vazão. O sistema operacional deve ser capaz de utilizar a CPU de forma eficiente, por meio de over-book[7] da largura de banda. Ex.: Vídeo sob demanda.
- Aplicações de melhor esforço (best-effort): São as aplicações convencionais que não necessitam de garantias de performance, requerendo apenas que a CPU seja alocada de forma que o tempo médio de resposta seja baixo e a vazão atingida seja alta. Ex.: Transferência remota de arquivos.

Os escalonadores de processos dos GPOS, contudo, são baseados em algoritmos de compartilhamento de tempo, como round robin (Tanenbaum, 1992), incapazes de oferecer quaisquer garantias sobre o tempo máximo de acesso à CPU (retardo máximo), por exemplo. Para a execução de processos em tempo real, o escalonador recorre ao uso de prioridades, o que pode levar à inanição aplicações de melhor esforço. Quando uma aplicação requer seu processamento em tempo

[7] O termo over-book, de origem inglesa, tem aqui o mesmo significado de quando é usado pelas companhias aéreas. Ele significa que as reservas para um vôo (CPU) contam com possíveis desistências (utilização da CPU não chega ao máximo), colocando mais passageiros (processos) na fila de espera do que realmente comportaria o avião.

real, a mais alta prioridade de execução é atribuída ao seu processo, que somente libera a CPU por vontade própria, ou para eventos do sistema com maior prioridade, ou durante a espera por uma operação de entrada ou saída. Esse esquema é injusto, já que apenas a aplicação em tempo real tem suas necessidades de processamento garantidas, enquanto pode ocorrer que outros processos não consigam a posse da CPU por um longo período de tempo.

Problemas com prioridades também são observados no funcionamento de subsistemas que utilizam interrupções para a transferência do processamento aos drivers de dispositivos ou a outras funções do *kernel*. Isso será visto com mais detalhes na descrição do subsistema de rede dos GPOS.

Dadas as diferentes necessidades das várias categorias de aplicações, nota-se que diferentes devem ser os algoritmos para escalonamento de processos. Aplicações em tempo real severo são bem atendidas por algoritmos de escalonamento baseados no tempo limite para execução, como Earliest Deadline First (EDF) e Rate Monotonic Algorithm (RMA) (Liu, 1973). Os processos da categoria tempo real suave devem ser escalonados por algoritmos que ofereçam certas garantias de QoS, mesmo na presença de sobrecarga, como o Start-time Fair Queuing (SFQ) (Goyal, 1996a). Já os algoritmos de compartilhamento de tempo, presentes nos GPOS, satisfazem as necessidades das aplicações de melhor esforço.

Os algoritmos de escalonamento de tempo real severo, como EDF e RMA, não são adequados para aplicações de tempo real suave, por não poderem admitir processos em over-book. Além disso, esses escalonadores exigem um conhecimento a priori do período e do tempo de execução, detalhes de difícil descrição para tais aplicações.

Assim, é necessário que o escalonamento de processos em sistemas operacionais permita a adaptação de escalonadores pelo sistema, conforme as necessidades das aplicações. Ao mesmo tempo, deve prover proteção entre as várias categorias de aplicação, facilitando a coexistência das políticas de escalonamento. Por exemplo, o overbooking da CPU para as aplicações de tempo real suave não deve violar as garantias dadas às aplicações de tempo real severo.

Da mesma forma, um comportamento anormal de aplicações de tempo real severo e suave não deve levar as aplicações de melhor esforço à inanição. Essa propriedade é também chamada de isolamento entre as categorias de escalonamento.

Nos GPOS, existem apenas as categorias de melhor esforço e, algumas vezes, de tempo real, que são regidas pelo mesmo algoritmo de escalonamento, com o uso de prioridades para os processos de tempo real. Dessa forma, os escalonadores dos GPOS não promovem um acesso justo à CPU para todos os processos em execução nem garante o isolamento entre as categorias.

2.2.3 Subsistema de rede

O subsistema de rede de sistemas operacionais é composto pelo conjunto de operações que manipulam o recebimento e envio de pacotes através de um enlace de rede. Para identificar as deficiências desse subsistema nos GPOS, é apresentada na Tabela 2.1 a seqüência de passos que compõem o processamento de rede sob o padrão 4.4BSD[8] (Wright, 1995). Na Figura 2.1 cada um dos passos está ilustrado ao longo da pilha de protocolos de rede.

Nota-se que o sistema pode ser analisado sob vários aspectos. Sob o foco da otimização de protocolos, os GPOS possuem um número elevado de leituras e cópias de dados ao longo da execução da pilha de protocolos. Tais operações devem ser evitadas por demandarem tempo de CPU para leitura e escrita e, ainda, espaço extra de memória para o armazenamento, no caso da cópia.

Com relação à provisão de QoS, estão listados a seguir os principais pontos críticos da arquitetura do subsistema de rede dos GPOS, alguns deles descritos em trabalhos como (Druschel, 1996), (Shi, 1998) e (Coulson, 1995):

1. Chamadas de sistema (1): Usadas pela aplicação tanto para a transmissão quanto para a recepção de dados, as chamadas de sistema

[8] O padrão BSD4.4 é utilizado em sistemas como UNIX e Linux. São pequenas as diferenças entre o padrão 4.4BSD e o subsistema de rede da família Windows (Microsoft, 1996).

constituem uma das formas de comunicação entre os níveis de usuário e de kernel. O tempo de CPU gasto pelo kernel é computado para o processo chamador, o que não provoca anomalias no escalonamento (o tempo de consumo da CPU é levado em consideração na decisão de escalonamento). O problema está no fato de que os GPOS não possibilitam a preempção do processamento do kernel, obrigando que outras operações somente sejam executadas quando o controle for devolvido ao processo chamador. Tal comportamento leva à inversão de prioridade, quando, por exemplo, um processo de maior prioridade que aquele que acionou a chamada de sistema encontra-se executável, requerendo o controle da CPU.

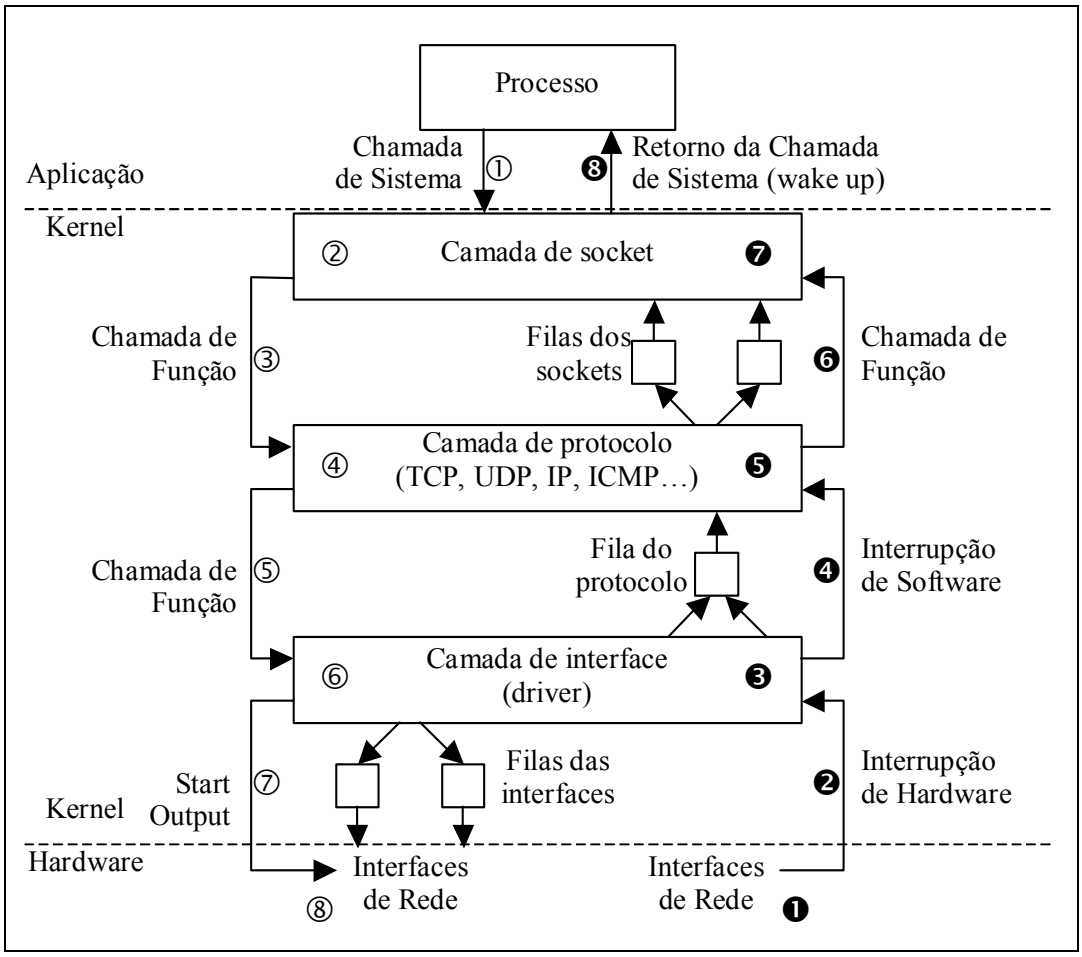

Figura 2.1 - Visão geral do processamento de rede no padrão BSD4.4

2. Chamadas de sistema (2): As chamadas de sistema dos GPOS bloqueiam a execução do processo e podem causar seu reescalonamento, se o sistema de comunicação não está pronto para o envio ou se os dados ainda não chegaram para a recepção. Observa-se, então, que existem

várias mudanças de contexto que poderiam ser evitadas se o processo envolvido na comunicação invocasse a chamada de forma assíncrona ou somente no momento em que a operação pudesse ser completada. Mudanças de contexto são caracterizadas por operações de armazenamento e recuperação dos registradores e tabelas de paginação do processo interrompido e do processo acordado, respectivamente. Tais operações podem representar perda de eficiência quando executadas com maior freqüência, principalmente por envolverem acesso à memória virtual, em alguns casos9. O suporte a chamadas de sistema assíncronas levaria a uma boa redução no número de mudanças de contexto.

Envio de dados	**Recepção de dados**
① O processo remetente solicita o envio dos dados através de uma **chamada de sistema**, cujos parâmetros são um apontador para o buffer de dados e o descritor do socket. ② O *kernel* copia os dados do buffer para um mbuf chain alocado de um pool. ③ Por uma chamada de função, é acionada a rotina de envio do protocolo correspondente ao tipo do socket informado. ④ O processamento da camada de protocolo envolve a adição de cabeçalhos aos mbufs, além de operações como checksum, que levam à leitura dos dados dos mbufs algumas vezes. ⑤ A camada de protocolo, após determinar a interface de saída, faz a chamada à rotina de envio do driver do dispositivo, passando o ponteiro para os mbufs. ⑥ O driver de dispositivo complementa o quadro de enlace e coloca os mbufs na fila de envio da interface. ⑦ Se a interface não estiver ocupada, o driver faz a chamada da função "start output" da interface diretamente, senão o mbuf ficará na fila até que os primeiros sejam enviados. ⑧ A interface, depois de processados os quadros predecessores na fila, copia os mbufs para seu buffer de transmissão e inicia o envio.	❶ A interface recebe os dados armazenando em seu buffer de recepção, até completar o quadro Ethernet. ❷ Ao completar o quadro, a interface dispara uma **interrupção de hardware**, que é tratada pelo *kernel*, escalonando o driver da interface. ❸ O driver do dispositivo copia os dados para um mbuf chain, alocado a partir do pool, retirando o cabeçalho da camada de enlace. ❹ O driver identifica o protocolo correspondente ao quadro, coloca os mbufs na fila do protocolo e dispara uma **interrupção de software**, tratada pelo *kernel* escalonando a recepção do protocolo para execução. ❺ A recepção na camada de protocolo envolve a retirada de cabeçalhos dos mbufs e outras operações que exigem leitura dos dados. ❻ O mbuf chain é colocado na fila do socket correspondente à porta local especificada. ❼ Quando o processo toma posse da CPU, a camada socket cria um mbuf identificando a origem dos dados. ❽ O processo receptor é acordado, pois estava bloqueado pela **chamada de sistema** de recepção. Quando escalonado para execução, os dados dos mbufs são copiados para o buffer do processo.

Tabela 2.1 - Descrição do processamento de rede no padrão BSD4.4

[9] Se as páginas de memória necessárias para o novo contexto de execução não mais se encontram na memória principal, devem ser recuperadas a partir da memória virtual.

3. Filas de pacotes: As filas de pacotes do tipo first come first served (FCFS) não garantem que os processos de alta prioridade tenham seus pacotes enviados com preferência sobre os de outros processos. De forma ideal, o escalonamento de pacotes de envio deve seguir a prioridade definida pelo escalonamento de processos ou aquela definida pela própria aplicação.

4. Utilização de interrupções de hardware: A manipulação de interrupções de hardware é uma das tarefas de mais alta prioridade em um sistema operacional, capaz de interromper qualquer processo de usuário, mesmo que ele não seja o responsável pela ação do dispositivo. Além disso, o tempo de CPU consumido no tratamento da interrupção é atribuído ao processo que foi interrompido, causando inversão de prioridade e anomalias no escalonamento, já que o tempo de CPU é contado como fator nos algoritmos de escalonamento.

5. Utilização de interrupções de software: A manipulação de interrupções de software está abaixo das interrupções de hardware na escala de prioridades de execução do sistema, mas acima da execução de processos de usuário. Isso leva aos mesmos problemas citados no item anterior, como a contabilidade inapropriada do tempo de uso dos recursos.

6. Descarte tardio de pacotes: O descarte de pacotes, como meio de diminuir a sobrecarga do receptor, pode ocorrer somente depois que muitos recursos de CPU foram investidos no tratamento do pacote descartado, levando a uma perda de capacidade de processamento da CPU.

2.3 Modelagem do sistema

A partir de um estudo sobre o funcionamento de sistemas operacionais, pode-se concluir que existem dois fluxos principais que comandam aplicações distribuídas:

- Fluxo de dados: é a seqüência composta por unidades de informação que são transmitidas entre threads. Essa transmissão pode envolver estações distintas, comunicando-se através de enlaces de rede, ou ocorrer internamente em uma mesma máquina, por meio de buffers ou parâmetros em chamadas de função (e.g. uma comunicação entre níveis de protocolo adjacentes).
- Fluxo de instruções: é a seqüência composta por comandos a serem interpretados por microprocessadores, que definem uma aplicação em execução ou uma rotina de controle do sistema, como uma entidade de protocolo.

Em sistemas operacionais multimídia, é necessário garantir ao fluxo de dados, alvo principal da provisão de QoS, que suas necessidades de utilização dos recursos de comunicação sejam respeitadas ao longo do caminho entre aplicação[10] e rede. Em várias etapas desse caminho, porém, é observada uma dependência entre o fluxo de dados e a capacidade de processamento de toda a pilha de protocolos, além da aplicação. Assim, o processamento do fluxo de instruções pela CPU deve ter garantias análogas àquelas dadas ao fluxo de dados, de forma que a interferência de um fluxo sobre o outro seja contabilizada para a gerência e controle da QoS como um todo.

Para visualizar esse relacionamento em um sistema operacional de uma forma genérica, considerando as arquiteturas aqui já descritas e as que serão apresentadas, é de grande utilidade a definição de um modelo de funcionamento do sistema. As técnicas de modelagem baseadas em redes de filas estendidas (Soares, 1990) podem ser utilizadas para descrever o comportamento dos fluxos de dados e de instruções em um sistema operacional, conforme ilustrado na Figura 2.2.

Algumas premissas foram adotadas para compor a generalidade do modelo. A principal delas é a definição de que cada nível da pilha de protocolos é executado por um processo em separado, dedicado a um certo fluxo de dados ou

[10] Neste ponto, o termo aplicação se refere a qualquer tarefa que envolve comunicação em rede, de aplicações do usuário a tarefas de controle do sistema, como o encaminhamento de pacotes.

compartilhado entre todos os fluxos. Não há perda de generalidade nessa premissa, já que se pode abstrair que uma entidade de protocolo reúna a execução de vários níveis reais da pilha, conforme for o espaço de endereçamento[11] em que trabalham esses níveis. Se não existe a mudança de espaço de endereçamento, não existe a necessidade de filas de comunicação entre esses níveis, pois os dados são passados entre as rotinas de tratamento por meio de chamadas de função.

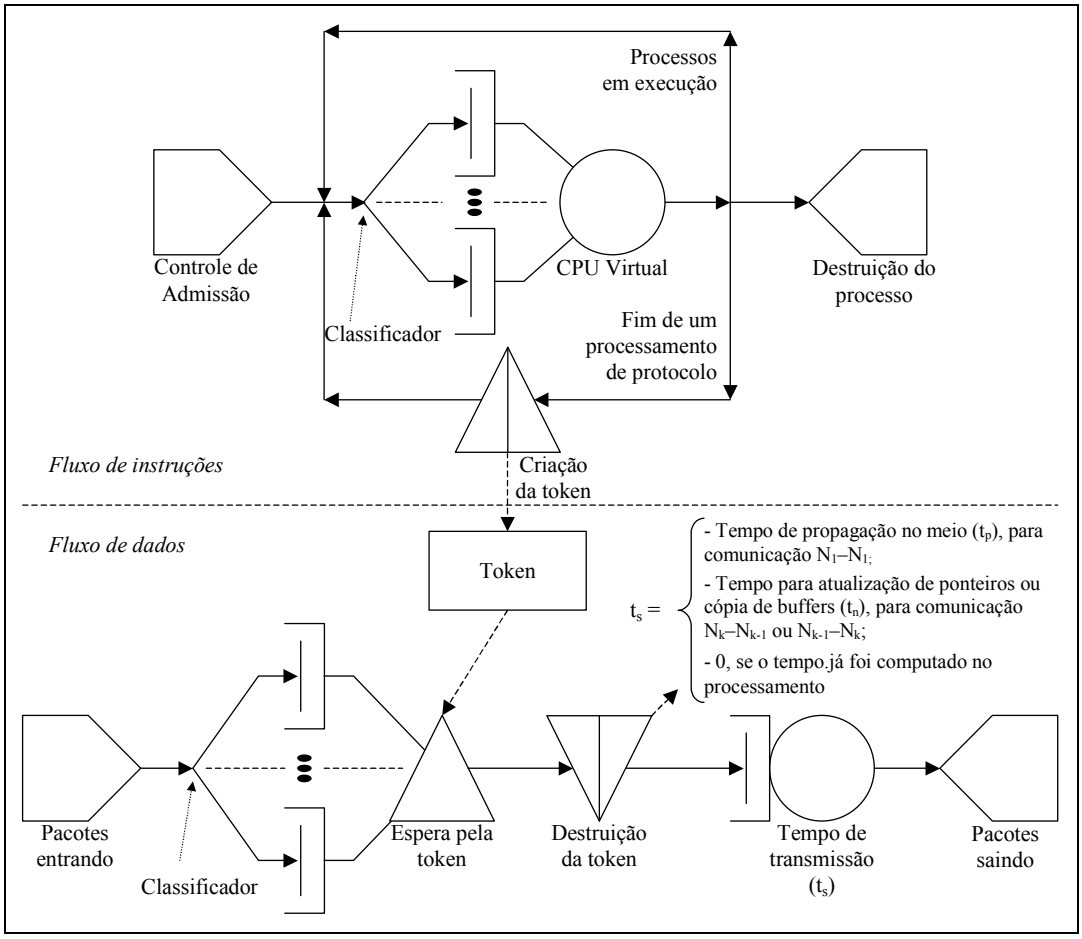

Figura 2.2 - Modelo baseado em redes de filas para sistemas operacionais em estações finais

Comparando-se alguns exemplos de arquiteturas, pode-se clarificar a descrição acima. Num GPOS, o *kernel* processa todas as camadas da pilha de protocolos em seu espaço de endereçamento, o que permite afirmar que tais camadas componham uma única entidade de protocolo, correspondendo ao contexto de um processo (não preemptivo, por ser, na realidade, o *kernel*).

[11] O espaço de endereçamento de um processo define o contexto e a área de memória exclusiva deste e de suas threads, onde podem ser feitas transferências de dados entre rotinas sem a necessidade de cópias.

Existem, portanto, filas de espera pelo processamento somente entre o hardware e o *kernel* e entre o *kernel* e a aplicação. Em sistemas em que a totalidade da pilha de protocolos está definida no espaço do usuário (Gopalakrishna, 1994), existirá um único conjunto de filas de pacotes em espera pelo processamento, entre o hardware e o processo que implementa a pilha de protocolos. Nesses sistemas, normalmente é o driver do dispositivo ou a própria interface que demultiplexa os pacotes de entrada entre as filas por socket. Não existem filas entre a aplicação e o processo do protocolo, já que eles podem estar no mesmo contexto ou, ainda, podem usar esquemas de memória compartilhada para o acesso aos dados.

Um processo em execução espera em uma das filas da CPU pelo seu escalonamento, que, quando efetuado, dá o direito de uso da CPU por um pequeno espaço de tempo. Terminado esse período, se o processo é uma entidade de protocolo e teve seu processamento completado, ele habilita o serviço de comunicação para a transferência dos dados entre os níveis correspondentes. Senão, ele volta para a fila de espera da CPU ou encerra sua execução.

O fluxo de dados, recebido do nível anterior (N_{k-1} ou N_{k+1}) ou adjacente (no caso N_1-N_1), aguarda a execução da entidade de protocolo correspondente, mantendo-se em uma fila de espera. Ao ser liberado, é transmitido para o próximo nível, totalizando um tempo de transmissão de acordo com a situação: se a comunicação é do tipo N_k-N_{k-1} ou N_{k-1}-N_k, o tempo será aquele necessário para a atualização de alguns ponteiros e/ou movimentação de buffers; se a comunicação é do tipo N_1-N_1, o tempo será igual ao tempo de propagação no meio.

Com esse modelo, ficam evidentes os principais recursos envolvidos na comunicação entre aplicações distribuídas e a relação de dependência entre os dois fluxos. O fluxo de dados somente é liberado quando o fluxo de instruções termina o processamento da entidade de protocolo correspondente, sendo que essa execução concorre com os demais processos existentes pelo uso da CPU.

Por exemplo, uma aplicação que exige alto desempenho para a exibição de um vídeo com boa performance deve competir pela CPU com as camadas de

protocolo que estão recebendo os seus próprios dados[12]. A coexistência de outras aplicações, como áudio, acirrará esta disputa, introduzindo também mais fluxos de dados que competirão pelas filas de espera e meios de transmissão, contra o fluxo de dados da aplicação de vídeo. Se tais recursos começarem a ficar escassos, ocorrerá a degradação da qualidade de algumas aplicações, evidenciando a importância da utilização de mecanismos para a provisão de qualidade de serviço sobre esses recursos.

Nota-se que esta importância não se dá apenas nas estações finais, mas também em alguns componentes do provedor de comunicações, como comutadores e roteadores. O mesmo tipo de relação é encontrado em tais recursos, onde um fluxo de instruções comanda a operação sobre o fluxo de dados.

2.4 Escalonamento de processos com qualidade de serviço

Nesta Seção, serão descritos trabalhos que visam preencher as deficiências encontradas nos GPOS quanto ao escalonamento de processos com QoS. Serão observados os mecanismos propostos com o objetivo de serem considerados na modelagem da arquitetura de frameworks aqui proposta.

2.4.1 Hierarchical Start-time Fair Queuing

O particionamento hierárquico da largura de banda da CPU (Goyal, 1996b) é uma das formas de suportar o uso de diferentes algoritmos de escalonamento por diferentes aplicações e prover o isolamento entre suas categorias. A hierarquia é representada por uma estrutura em árvore, onde cada thread[13] pertence a exatamente um nó folha. Cada nó folha representa um agregado de threads e,

[12] Nota-se que nos GPOS, como já comentado, não existe a concorrência entre o processo que envia os dados com o seu processamento de protocolos. Porém, a recepção guiada por interrupções pode provocar a preempção de qualquer processo mesmo que não relacionado aos dados sendo recebidos.

[13] O termo thread é muito difundido nos trabalhos sobre sistemas operacionais, mesmo em línguas diferentes do inglês, dada a dificuldade de tradução direta. Ao longo deste trabalho o termo thread será utilizado para representar qualquer entidade de processamento escalonável.

portanto, uma categoria de aplicações. Cada nó interno na árvore representa um agregado de categorias de aplicações. Todos os nós possuem um peso que determina a percentagem de largura de banda que deve ser alocada para a classe de aplicações representada por esse nó, a partir de seu nó pai. Além disso, cada nó possui um escalonador: o escalonador de um nó folha escalona todas as threads pertencentes ao nó folha; o escalonador de um nó intermediário escalona seus nós filhos.

A Figura 2.3 ilustra uma possível estrutura de escalonamento. Nela, a classe root tem três subclasses: tempo real severo, temo real suave e melhor esforço, com os pesos 1, 3 e 6, respectivamente. A largura de banda da classe melhor esforço foi dividida igualmente entre as classes folhas usuário1 e usuário2, com pesos iguais a 1. Esta distribuição de pesos equivale à descrição de que 10% da largura de banda da CPU será destinada à categoria de aplicações de tempo real severo, 30% à categoria de tempo real suave e 60% à classe de aplicações de melhor esforço; desses 60%, 50% estão reservados para as threads de usuário1 e 50% para as threads pertencentes a usuário2. Enquanto as classes de tempo real suave e usuário1 executam um algoritmo de escalonamento de distribuição justa, as classes tempo real severo e usuário2 utilizam os escalonadores EDF e de tempo compartilhado, respectivamente.

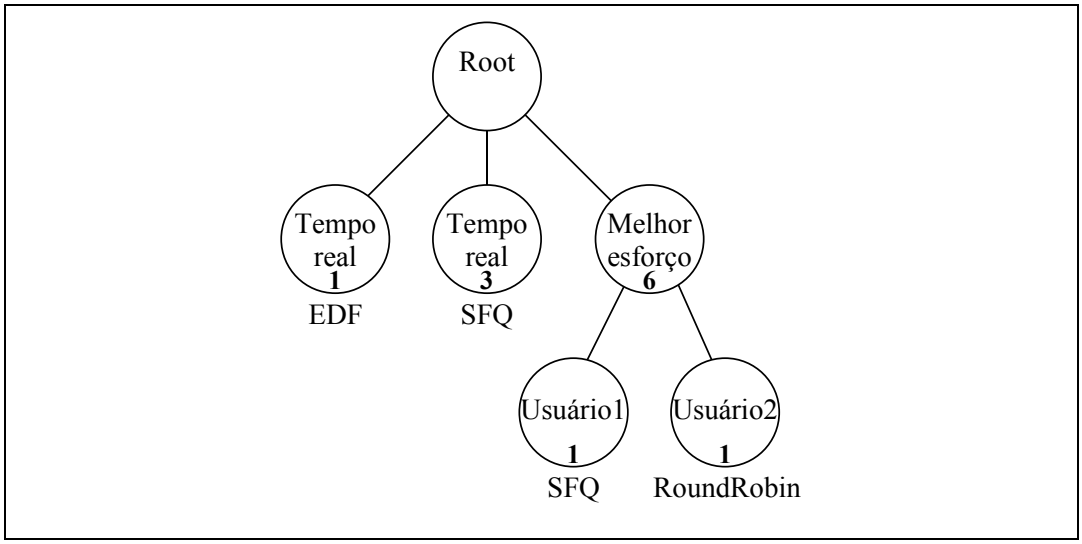

Figura 2.3 - Exemplo de estrutura de escalonamento

Observa-se, então, que os escalonadores dos nós folhas da hierarquia são determinados de acordo com as necessidades das aplicações. Para o

escalonamento dos nós intermediários foi proposto o algoritmo SFQ, responsável por garantir a total justiça do uso da CPU entre as classes de aplicações. Em sistemas com alto grau de adaptabilidade, porém, pode ser desejável a definição de um novo escalonador também para nós intermediários em alguns ramos da árvore. O uso do SFQ se tornaria apenas um caso particular em uma arquitetura realmente adaptável como a proposta no presente trabalho.

(Goyal, 1996b) propõe, ainda, que a infra-estrutura de escalonamento seja usada por um *gerenciador de QoS* (Figura 2.4), para o qual as aplicações irão direcionar suas *solicitações*, especificando suas necessidades. O gerenciador de QoS deve ser capaz de determinar os recursos necessários para atingir as requisições de QoS das aplicações e decidir a qual *classe de escalonamento* a aplicação deve pertencer. O mesmo gerenciador poderá executar procedimentos de *controle de admissão*, para determinar se as requisições por recursos podem ser satisfeitas e *alocar os recursos* para a aplicação na classe apropriada.

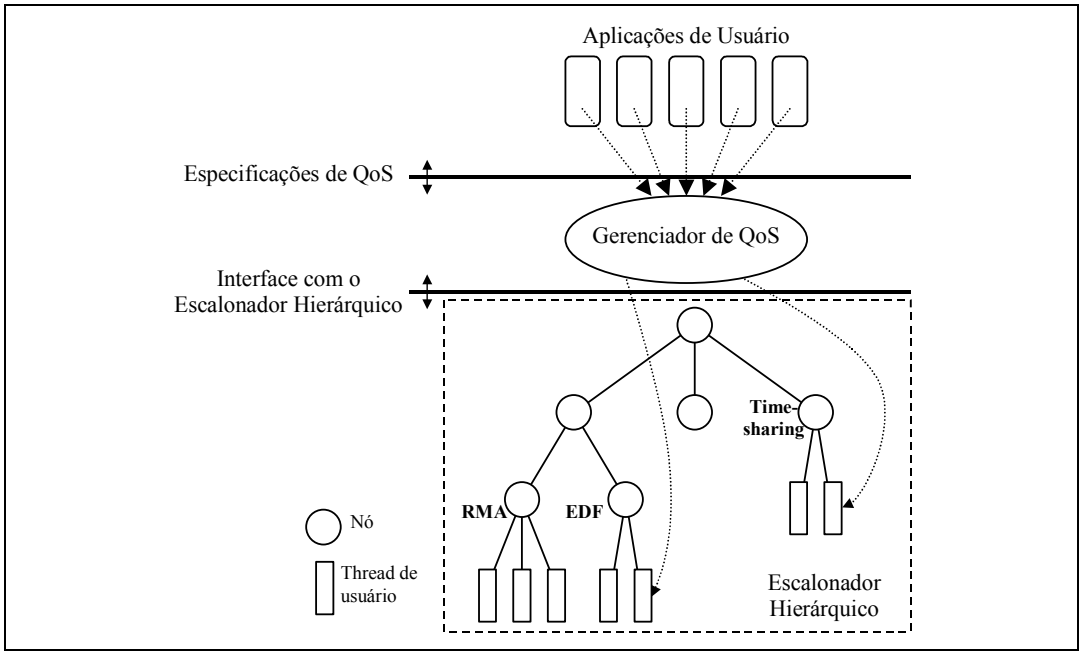

Figura 2.4 - Interação do Gerenciador de QoS com o escalonador hierárquico

Além dessas tarefas, o gerenciador pode mover as aplicações entre as classes ou modificar a alocação de recursos em resposta a uma mudança nas necessidades de QoS ou na situação de carga (*renegociação e sintonização de QoS*). O trabalho citado não abrangeu a programação do gerenciador de QoS, deixando em aberto as formas de implementação dessas políticas e o conjunto de

parâmetros de QoS das requisições. A arquitetura proposta pelo presente trabalho inclui uma discussão detalhada sobre esse conjunto de mecanismos, apresentada no Capítulo 3.

2.4.2 Outros escalonadores hierárquicos

Em (Ford, 1996), é proposto um escalonamento de CPU por herança (CPU Inheritance Scheduling), um framework no qual threads arbitrárias podem agir como escalonadores para outras threads. Uma thread pode, então, "doar temporariamente" seu tempo de CPU para as threads selecionadas, enquanto esperam por eventos de seu interesse, como interrupções de clock ou timer. A thread que recebe este tempo de CPU pode passá-lo para outras threads, formando uma hierarquia lógica de escalonadores.

(Regehr, 2001) propõe uma infra-estrutura de escalonadores hierárquicos carregáveis em tempo de execução (Hierarchical Loadable Scheduler), provendo uma interface de programação bem definida para a construção de escalonadores. Esses, por sua vez, são notificados sobre todos os eventos do sistema operacional que podem demandar decisões de escalonamento, devendo estar aptos a tomar a ação apropriada. Para isso, foi definido um modelo de programação de escalonadores carregáveis, formalizando as suas possíveis interações com o *kernel*. O modelo inclui quando e por que diferentes notificações são enviadas aos escalonadores, quais as ações que eles podem tomar ao respondê-las, como é feito o controle de concorrência na infra-estrutura e propõe um modelo de confiança para os escalonadores carregáveis. Um protótipo foi criado sobre o *kernel* do sistema operacional Windows 2000.

A estrutura de escalonamento hierárquico será apresentada de forma genérica no presente trabalho, de forma a abranger outros recursos do sistema, além da CPU, através do conceito de *árvores de recursos virtuais*, introduzido no Capítulo 3.

2.4.3
Meta-algoritmo para Políticas de Escalonamento

O meta-algoritmo LDS – Load Dependent Scheduling (Barria, 2001) foi desenvolvido com o objetivo de se tornar uma ferramenta para a análise e comparação do desempenho de diversos algoritmos de escalonamento. Ele é capaz de emular a operação de vários algoritmos, a partir da modificação dos seus próprios parâmetros de operação. A implementação do LDS para o escalonamento de recursos seria vantajosa para a provisão de QoS, já que vários algoritmos que oferecem certas garantias de processamento podem ser por ele emulados (e.g. weighted fair queuing (Demers, 1990) – WFQ).

Além disso, o LDS introduz mais um ponto de *adaptabilidade* ao sistema, onde a adição de novas estratégias de escalonamento pode ser feita simplesmente através da mudança dos valores que controlam a operação do algoritmo. Nota-se que esse esquema de adaptação possui uma vantagem sobre a inserção de módulos no *kernel*, por não exigir a execução de etapas anteriores para a implementação das estratégias, como programação e geração de código objeto.

Por outro lado, o LDS não possibilita a utilização simultânea de mais de uma política de escalonamento, o que impede o atendimento satisfatório a múltiplas categorias de aplicações. Além disso, o algoritmo não é capaz de emular políticas que envolvem a determinação de prazos limites para execução, como os algoritmos de escalonamento para aplicações de tempo real severo. Um último problema pode estar no grande número de entradas da tabela para caracterizar todas as possíveis situações de carga de recursos como a CPU.

A utilização do algoritmo LDS em uma estrutura hierárquica de escalonamento é uma proposta que possibilitaria a criação de diferentes estratégias para as categorias de aplicações. Da mesma forma, o número de entradas na tabela seria reduzido, já que existiria uma tabela para cada categoria, caracterizando uma situação local de carga. A deficiência para emulação de algoritmos de tempo real pode ser suprida pelo uso do LDS nos escalonadores internos da hierarquia, deixando para os escalonadores folhas a implementação de algoritmos mais específicos, quando houver essa necessidade.

O LDS opera ciclicamente (Figura 2.5), onde cada ciclo é composto por *N+1* fases (*N* corresponde ao número de filas de execução da CPU). A fase f_0 corresponde a uma fase de inatividade. Durante a fase f_i ($1 \leq i \leq N$), a CPU atenderá exclusivamente as threads da fila *i*, sendo que a duração da fase depende do estado que caracteriza a carga da CPU. Assim, o tamanho do quantum ou, se preferível, o número de quanta que serão atribuídos à execução de cada fase (threads de uma mesma categoria) pode variar em diferentes ciclos de operação do LDS.

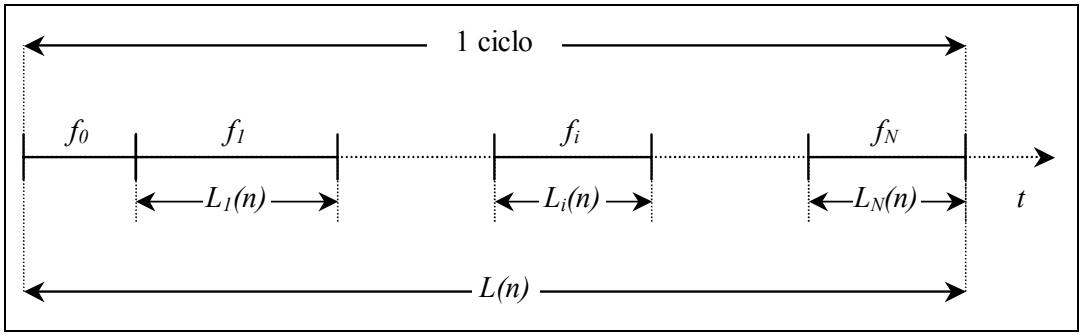

Figura 2.5 - Diagrama temporal de um ciclo de execução do algoritmo LDS

Os dados que compõem a situação de carga da CPU (número de threads em execução em cada fila da CPU) indexam uma tabela para a localização do vetor que descreverá a duração de execução de cada thread no ciclo sendo iniciado. A tabela LDS (Tabela 2.2), como é chamada, possui *2N* colunas, onde *N* corresponde à quantidade de filas atendidas pela CPU. As primeiras *N* colunas são rotuladas por n_i, indicando o número de threads em execução em cada fila da CPU no início do ciclo. As *N* colunas restantes são rotuladas com $L_i(n)$, indicando o número máximo de quanta que pode ser atribuído para a execução da fase f_i. A quantidade de linhas da tabela é igual ao número de combinações lógicas diferentes que podem caracterizar a carga da CPU.

N_1	N_2	N_3		N_N	$L_1(n)$	$L_2(n)$	$L_3(n)$		$L_N(n)$

Tabela 2.2 - Cabeçalho da Tabela LDS

O algoritmo LDS pode ser descrito em alto nível como a seguinte seqüência de passos:

1. No início de um ciclo, é determinado o vetor de estado n, caracterizando a carga da CPU.

2. Se n=0, deve-se colocar a CPU em estado ocioso, esperando o início da execução de alguma thread (fase f0). Quando for iniciada a execução de alguma thread, inicia-se a fase de atendimento àquela que chegar primeiro.

3. Se n≠0, são lidas as N primeiras colunas de cada linha da tabela LDS, até que se encontre a situação de carga atual da CPU. Encontrada a linha correspondente, são lidas as próximas N colunas, especificando o valor de Li(n).

4. A seguir, é dado o controle da CPU às threads da fase f1, por tantos quanta quantos os especificados por L1(n). Em seguida, são atendidas as threads da fase f2, e assim por diante, até a fase fN.

5. Durante a fase fi, as threads da fila i são atendidas pela CPU de forma exclusiva, pelo período determinado pelo número de quanta Li(n). Se durante essa fase as threads entram em estado bloqueado ou encerram, a fase fi deve ser abortada para o início da fase fi+1.

2.5 Subsistema de rede com qualidade de serviço

Para buscar a solução dos problemas encontrados no subsistema de rede dos GPOS, muitos trabalhos definem a criação de novos sistemas operacionais baseados, principalmente, na arquitetura de *microkernel*.

Nessa alternativa aos sistemas monolíticos, o *kernel* do sistema é muito pequeno, responsável por comandar a passagem de mensagens entre os módulos do sistema e o hardware, além de funções básicas como trocas de contexto entre processos. Algumas funções, como escalonamento de processos e controle de entrada e saída, podem estar implementadas no núcleo, mas a idéia é mantê-lo com o menor tamanho possível. A tradução para uma outra arquitetura de hardware envolveria apenas a modificação código do *microkernel*, já que os módulos não têm dependência direta do hardware, facilitando a portabilidade do sistema. A adaptabilidade a novos serviços é beneficiada, já que um módulo do sistema é, geralmente, implementado por um processo que se comunica com o

microkernel, bastando para a sua substituição o encerramento do processo antigo e o disparo do novo. Outra vantagem observada está no carregamento apenas dos módulos necessários no momento, sem a ativação daqueles que não serão utilizados, representando uma boa economia de memória e processamento.

Uma das críticas ao *microkernel* é a baixa velocidade na comunicação entre os módulos, por ser baseada em troca de mensagens. A arquitetura de passagem de mensagens tem sua eficiência comprometida principalmente pela sobrecarga de mensagens geradas. Questões de segurança na substituição e inserção de módulos também devem ser observadas, já que um módulo mal escrito ou programado por um usuário mal intencionado pode afetar a estabilidade do sistema. Finalmente, os sistemas baseados em *microkernel* não exploram funções específicas do hardware por definir seus drivers em módulos que não se relacionam diretamente a ele. Essas funções podem incluir alternativas de acesso direto ao dispositivo, ou instruções especiais que representam um incremento no desempenho desse acesso.

Nesta Seção, serão apresentados alguns sistemas baseados em *microkernel* para a provisão de QoS com ênfase no subsistema de rede, sendo que alguns deles definem, também, soluções para o escalonamento de processos. Outros trabalhos que serão descritos visam a solução de problemas isolados no processamento da pilha de protocolos de rede dos GPOS.

2.5.1 SUMO

O projeto SUMO (Support for Multimedia in Operating Systems) (Coulson, 1995) da Universidade de Lancaster engloba a implementação de um sistema operacional baseado em um *microkernel* – Chorus (Campbell, 1996) – oferecendo facilidades para o suporte a aplicações multimídia distribuídas. A arquitetura do sistema é baseada em três novos princípios:

- As aplicações são guiadas por “upcalls”, definindo que os eventos de comunicação são iniciados pelo sistema e não pela aplicação;

- As funções de gerenciamento de recursos são executadas cooperativamente entre o *kernel* e componentes do nível de usuário (sistema split-level);
- A transferência de controle é feita separadamente da transferência de dados, de forma assíncrona.

Além disso, estruturas e mecanismos para o gerenciamento de recursos com QoS são definidos. A seguir, serão abordadas essas características.

Aplicações guiadas por upcalls

A infra-estrutura do sistema SUMO foi projetada para que trabalhasse de forma ativa, deixando as aplicações com um processamento passivo. Isso significa que, no estabelecimento de uma conexão, o sistema, e não a aplicação, dispara as threads responsáveis por tratar a comunicação e aloca os buffers para as conexões. No momento da transferência de dados, é o sistema quem decide por ativar a aplicação através de uma chamada (upcall), nos instantes determinados pela *especificação de QoS* fornecida pela aplicação.

Neste tipo de interação entre *kernel* e aplicação, como o sistema realiza o escalonamento do processo de acordo com a comunicação, a *monitoração* e o *gerenciamento de QoS* para a conexão são providos com maior facilidade. Por exemplo, não existe a necessidade de policiamento e moldagem do fluxo gerado pela aplicação, pois esta não consegue enviar dados fora do tempo concedido pela sua especificação de QoS. Além disso, é reduzido o número de mudanças de contexto, pois o processo é ativado quando existem os dados a serem recebidos ou quando os dados por ele gerados podem ser enviados. Outra vantagem da arquitetura está no modelo de programação orientado a eventos em que é baseada, por ser ideal para a estruturação de aplicações distribuídas de tempo real. O programador declara como os eventos de comunicação serão tratados, mas o momento de execução deste tratamento é governado pelos parâmetros de QoS fornecidos em tempo de conexão.

Sistema split-level

A estruturação split-level define que as funções-chave do sistema devem ter seu desempenho compartilhado cooperativamente entre o kernel e o nível de usuário, intercomunicando-se de forma assíncrona para a troca de informações de gerenciamento. Assim, tira-se proveito da minimização de overheads de comunicação conseguida através da implementação de várias funções do sistema no mesmo espaço de endereçamento da aplicação.

No escalonamento de processos split-level (Govindan, 1991), um pequeno número de processadores virtuais executa threads de usuário em cada espaço de endereçamento, seguindo as definições: i) cada escalonador de nível de usuário (user-level scheduler – ULS) sempre escolhe a thread de usuário mais urgente; ii) cada escalonador de nível de *kernel* (kernel-level scheduler – KLS) sempre escolhe o processador virtual que possui a thread de usuário mais urgente no contexto geral. A arquitetura permite, dessa forma, mudanças de contexto baratas entre threads no mesmo espaço de endereçamento, ao mesmo tempo em que as urgências relativas entre threads pertencentes a outros espaços de endereçamento são asseguradas.

Na comunicação split-level, o *kernel* fica responsável por multiplexar e demultiplexar os pacotes de rede entre os espaços de endereçamento das aplicações, deixando que cada uma delas realize o processamento do nível de transporte. O processamento da comunicação pode, então, tirar vantagem do escalonamento split-level, pela implementação das threads de comunicação no mesmo espaço de endereçamento da aplicação. Dessa maneira, as mudanças de contexto entre threads de processamento e de comunicação terão baixo custo, não haverá a necessidade de cópias de dados e o processamento da comunicação poderá seguir a urgência da aplicação correspondente.

Separação da transferência de controle da transferência de dados

Nos GPOS, as transferências de controle e de dados estão acopladas, como no caso de uma chamada de sistema em que os dados são passados para o *kernel* ao mesmo tempo em que o controle de execução é transferido. No SUMO, existe

a separação da transferência de controle da transferência de dados, facilitando a implementação de chamadas de sistema e interrupções de software assíncronas, por exemplo. A definição de chamadas de sistema assíncronas é interessante por reduzir, também, o número de mudanças de contexto provocadas por processos que antes seriam bloqueados na espera por uma resposta às suas requisições.

Gerenciamento de recursos

(Campbell, 1996) define dois estágios que antecedem uma efetiva reserva de recursos em sistemas operacionais. O *mapeamento de QoS* é o processo de transformação dos parâmetros descritos no contrato de serviço (*especificação de QoS*) em valores que representam as reais necessidades de desempenho para cada um dos recursos envolvidos. Por exemplo, o subsistema de rede pode oferecer garantias sobre a largura de banda, retardo máximo e taxa de perda de pacotes. Esses valores são calculados a partir dos parâmetros informados na especificação de QoS. Para uma aplicação de vídeo, a largura de banda pode ser calculada a partir da taxa de quadros de vídeo e do tamanho do quadro. O retardo máximo é obtido através da parcela atribuída à estação pelo protocolo de negociação, proveniente da distribuição da latência total informada na especificação. A taxa de perda de pacotes é calculada a partir de parâmetros de mais alto nível como perda de quadros e intervalo entre perdas.

A *verificação de admissão* determina se recursos suficientes para atender aos parâmetros descritos acima estão disponíveis no sistema. Por exemplo, o subsistema de rede de cada nó que define o caminho do fluxo a ser admitido deve realizar três diferentes verificações: teste de largura de banda; teste de retardo máximo e teste de disponibilidade de buffers.

O gerenciador de fluxos ilustrado na Figura 2.6 é uma entidade responsável por administrar as necessidades locais no sistema operacional para os fluxos de rede, distribuindo a responsabilidade da gerência de QoS entre os módulos de gerenciamento individuais de recursos (CPU, memória e rede). O gerenciador de fluxos utiliza um protocolo de gerenciamento de fluxos para obter a parcela da QoS especificada que deve por ele ser negociada localmente.

Dessa forma, quando uma conexão com QoS é solicitada, a função de mapeamento traduz os parâmetros de nível do usuário em parâmetros a serem considerados por cada um dos gerenciadores de recursos, inclusive o gerenciador de fluxos. Com base na distribuição da alocação de recursos entre os nós participantes da comunicação, o gerenciador de fluxos interage com os gerenciadores de CPU, memória e rede para que sejam feitas as verificações de admissão, de forma independente. Se todos eles responderem positivamente sobre a viabilidade da reserva requerida, a conexão pode ser aceita, caso contrário pode ser feita uma renegociação. O gerenciador de fluxos é, ainda, responsável pelo ajuste dinâmico das reservas de recursos (*sintonização de QoS*), em resposta a ocorrências de degradação da QoS no momento da provisão do serviço.

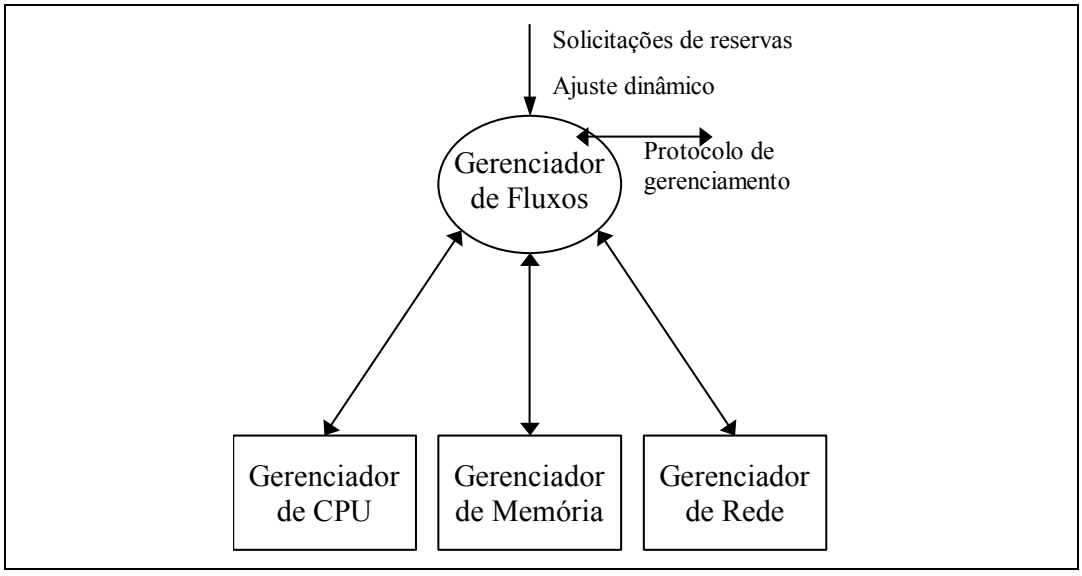

Figura 2.6 - Gerenciador de fluxos proposto por (Campbell, 1996)

2.5.2 Nemesis

O projeto Nemesis (Leslie, 1996) da Cambridge University é um exemplo de sistema operacional estruturado verticalmente (Black, 1997), onde as aplicações realizam a maior parte possível do processamento a elas pertinente, ao invés de passar para o *kernel* ou para servidores esse trabalho. As APIs que implementam as funções do sistema operacional são disponibilizadas como bibliotecas compartilhadas. Existe apenas um espaço de endereçamento virtual, sem perda da proteção de memória entre os domínios de aplicação (processos).

Esse espaço único reduz o número de mudanças de contexto, claramente vantajoso para aplicações com fortes requisitos de QoS, como já visto.

O *microkernel* Nemesis consiste apenas do escalonador e de manipuladores de interrupções e traps, não havendo threads de *kernel*. A Figura 2.7 mostra a organização do Nemesis como um conjunto de domínios e um pequeno *kernel*. O projeto considera que as interfaces de rede são capazes de demultiplexar os dados recebidos, colocando-os diretamente na fila de destino correspondente. A aplicação contém um protocolo estruturado verticalmente para cada fluxo de comunicação, estabelecendo um mesmo espaço de endereçamento para aplicação e processamento de protocolos.

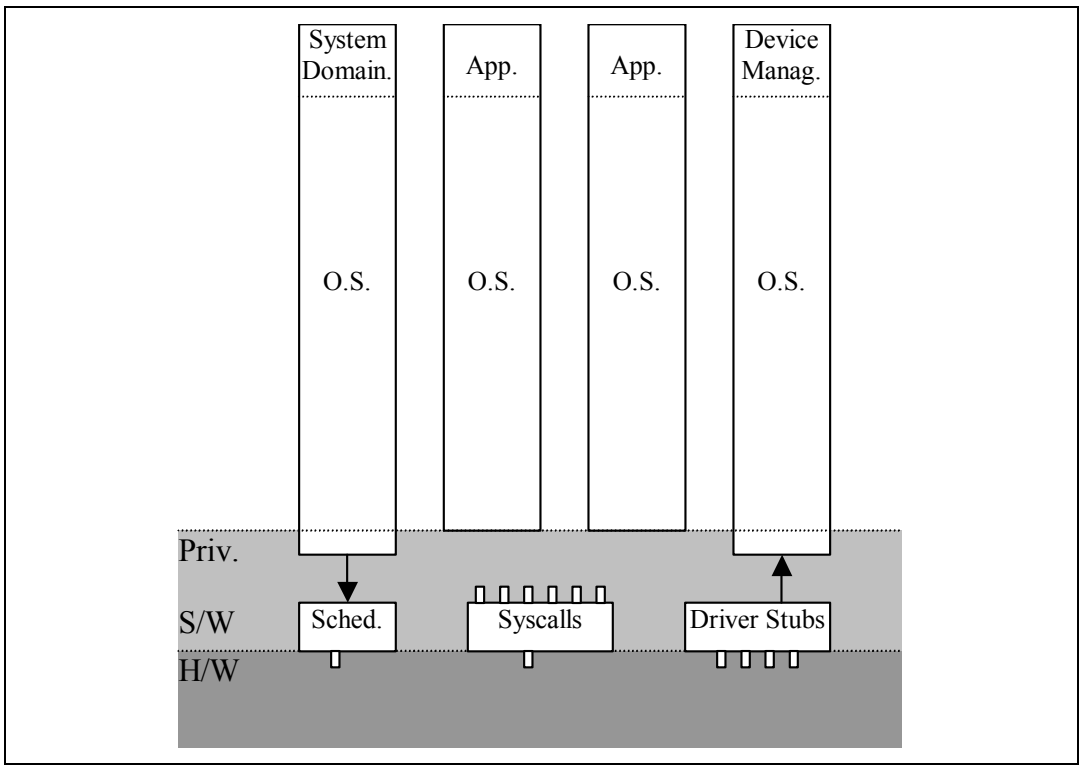

Figura 2.7 - Arquitetura do sistema Nemesis

O sistema Nemesis possui uma arquitetura de gerenciamento de recursos (Oparah, 1998) cujo elemento central é denominado Gerenciador de QoS, representado por um serviço do sistema e responsável por coordenar a distribuição dos recursos entre os domínios de aplicação. A fim de obter garantias de QoS ou renegociar uma garantia antiga, as aplicações devem encaminhar uma requisição ao gerenciador de QoS que, baseado na disponibilidade dos recursos, distribui suas decisões de alocação entre os gerenciadores de recursos. Cada recurso do

sistema operacional, como CPU, buffers de comunicação e disco, possui um gerenciador de recurso designado para assegurar que as garantias solicitadas sejam respeitadas pelo escalonamento apropriado do recurso.

O gerenciador de QoS provê, também, uma interface gráfica para o usuário, para que o estado atual de alocação de recursos para cada uma das aplicações possa ser informado e ajustado. A arquitetura foi implementada através de um protótipo que oferece somente o gerenciamento da CPU. Nesse caso, o usuário pode redefinir certas garantias de alocação da CPU, como a percentagem reservada para uma dada aplicação. A Figura 2.8 ilustra de forma simplificada a arquitetura de gerenciamento de recursos do Nemesis.

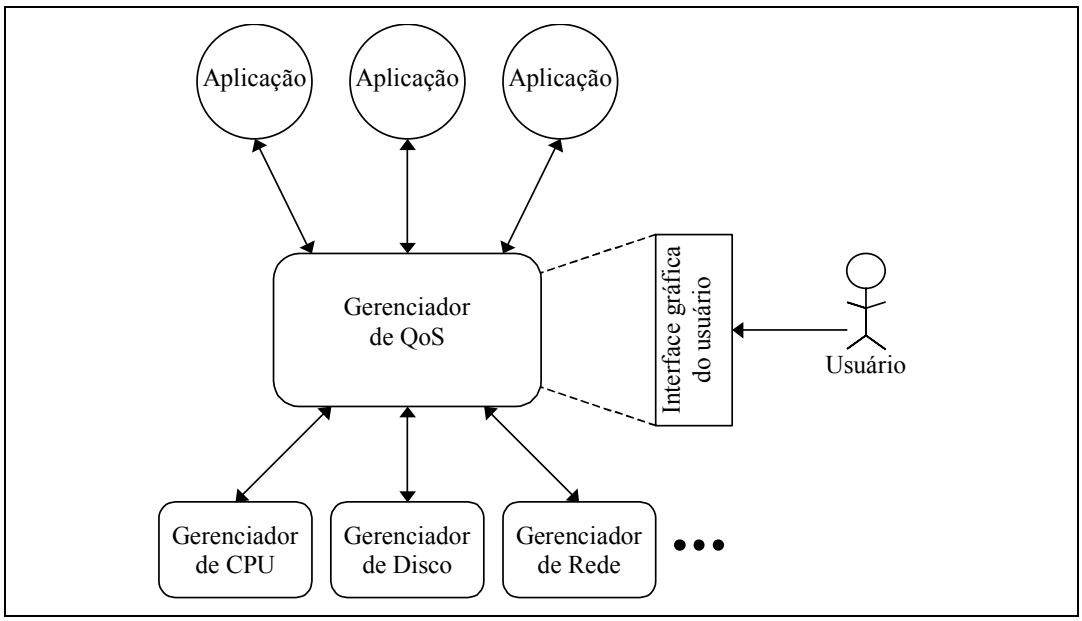

Figura 2.8 - Arquitetura de gerenciamento de recursos do Nemesis

2.5.3 Real Time Mach

O sistema RT-Mach (Mercer, 1994) (Lee, 1996) é uma extensão ao *kernel* Mach para o suporte a aplicações de tempo real. O antigo subsistema de rede baseado em um servidor que manipula as chamadas baseadas no padrão BSD foi substituído por uma arquitetura de processamento de protocolo no nível de usuário.

O servidor UX utilizado no Mach não preenchia as necessidades de processamento em tempo real por não suportar qualquer tipo de prioridade. Além disso, a comunicação feita entre a aplicação e esse servidor adicionava um overhead, diminuindo a vazão e aumentando a latência. Os princípios que guiaram os pesquisadores para o desenvolvimento de um subsistema de rede preditivo foram os seguintes:

1. Utilização de prioridades nos pacotes para enfileiramento;
2. Escalonamento do processamento de protocolo contra as outras atividades do sistema utilizando a prioridade dos pacotes;
3. Utilização de uma estrutura de controle de preempção para reduzir a ocorrência de inversão de prioridade e mudanças de contexto;
4. Particionamento dos recursos para eliminar a interferência entre classes de prioridade;

Assim, o grupo criou uma biblioteca que manipula o processamento de protocolo para o envio e recebimento de pacotes, interagindo com os drivers de filtragem de pacotes e de interface de rede diretamente. A biblioteca pode ser linkeditada às aplicações, de forma que elas possam fazer seu próprio processamento de protocolo em seu espaço de endereçamento. A biblioteca somente interage com o servidor UX para criar e destruir conexões e outras poucas operações.

2.5.4 Process per Channel

(Mehra, 1996) propõe uma arquitetura para o subsistema de comunicação baseado no modelo processo por canal, no qual cada canal é coordenado por um manipulador único e exclusivo, implementado como um processo leve (lightweight process - LWP), criado no estabelecimento do canal.

O envio de uma mensagem é feito através de uma API. Depois de feita a moldagem do tráfego, a mensagem é enfileirada na fila de mensagens do canal, para o subseqüente processamento pelo manipulador do canal. De acordo com o

tipo de canal, o manipulador é associado a uma das três filas de execução da CPU (current real time: processando mensagens que obedecem à taxa do canal; early real time: processando mensagens que violam a taxa do canal; e best-effort: canais de melhor esforço). Quando escalonado para execução (estratégia EDF), o manipulador retira cada mensagem para o processamento do protocolo, podendo ser interrompido por threads de maior prioridade. Os pacotes gerados pela segmentação da mensagem são inseridos em uma das três filas de pacotes de enlace da interface correspondente, sendo posteriormente transmitidos, de acordo com a estratégia do escalonador da interface.

Na recepção, o pacote recebido é demultiplexado diretamente para a fila de pacotes do canal correspondente, para o processamento e remontagem. O manipulador do canal é associado a uma das filas de execução da CPU e, quando escalonado, processa os pacotes da fila, podendo ser interrompido para preempção. Chegado o último pacote que completa a remontagem da mensagem, o manipulador a coloca na fila de mensagens do canal, sendo retirada quando a aplicação fizer a chamada de recepção da API.

2.5.5 Lazy Receiver Processing (LRP)

Na arquitetura proposta em (Druschel, 1996), chamada Lazy Receiver Processing (LRP) – Processamento tardio do receptor – alguns dos problemas de *recepção de dados* do subsistema de rede dos GPOS são resolvidos pela seguinte combinação de técnicas:

1. Substituição da fila compartilhada IP por filas por socket, de acesso direto pela interface de rede. A interface deve, então, demultiplexar os pacotes de entrada colocando-os nas filas apropriadas, de acordo com o socket de destino.
2. Execução do protocolo de recebimento executado na prioridade do processo receptor. Para isso, o processamento será executado mais tarde, no contexto da chamada de sistema disparada pelo processo responsável pelo recebimento.

Assim, o processamento do protocolo para um pacote não ocorre até que a aplicação receptora requisite-o pela chamada de sistema. Além disso, esse processamento não mais interrompe o processo em execução no momento da chegada do pacote, a não ser que o receptor tenha maior prioridade no escalonamento que o processo corrente. Evitam-se, ainda, mudanças de contexto inapropriadas, levando a um significativo aumento do desempenho do sistema.

No LRP, a própria interface de rede (seja através de firmware ou pelo driver) separa o tráfego chegando para o socket de destino e coloca os pacotes diretamente nas filas de recebimento, exclusivas de cada socket. Combinado com o processamento da pilha de protocolos na prioridade da aplicação receptora, tem-se um mecanismo de feedback para a interface sobre a capacidade das aplicações de manter o trabalho com o tráfego de entrada de um socket. Essa informação pode ser usada de forma que, quando a fila de um socket se esgota, a interface de rede possa descartar os próximos pacotes a ele destinados até que a aplicação consuma algumas posições da fila. Desse modo, a interface pode fazer o descarte sem que recursos excessivos sejam comprometidos no processamento desses pacotes a serem perdidos.

A separação do tráfego recebido pela interface de rede, combinada com o processamento na prioridade da aplicação, elimina interferências entre os pacotes destinados para sockets distintos. Adicionalmente, a latência de despacho de um pacote não é influenciada pelo subseqüente recebimento de um pacote de prioridade menor ou igual. A eliminação da fila IP compartilhada reduz fortemente a probabilidade de o pacote ser retardado ou descartado devido à ausência de recursos provocada por um tráfego destinado a um socket diferente.

A arquitetura define que o tempo de CPU gasto no processamento do protocolo de recebimento (excluindo-se o tempo para manipulação da interrupção de hardware) é contabilizado para o processo que recebeu o tráfego, fato importante já que o uso recente de CPU influencia na prioridade de escalonamento dos processos. Garante-se, então, melhor justiça no caso em que os processos de baixa prioridade recebem grandes volumes de tráfego de rede.

2.5.6
Linux Network Traffic Control (LinuxTC)

As versões mais recentes de *kernel* do Linux oferecem um grande conjunto de funções de controle de tráfego de rede (Almesberger, 1999) (Radhakrishnan, 1999). As estruturas utilizadas para isso são capazes de oferecer os mecanismos necessários para o suporte às arquiteturas IntServ (Braden, 1994) e DiffServ (Blake, 1998).

De maneira geral, pode-se descrever o processamento feito pelo subsistema de rede do *kernel* do Linux, a partir da Figura 2.9. Um pacote recebido pela interface de rede é examinado, para que o *kernel* decida entre o encaminhamento para outro nó da rede e o processamento pelos protocolos de níveis mais altos da pilha. No caso de roteadores, a maioria dos pacotes será analisada para o posterior encaminhamento em uma de suas interfaces de rede. Para estações finais (hosts), os pacotes serão processados por um protocolo de transporte, tal qual UDP ou TCP. Essas camadas mais altas da pilha podem, também, gerar dados para as camadas mais baixas, para as tarefas de transmissão de dados, roteamento e encapsulamento. O encaminhamento inclui a seleção da interface de saída, a escolha do próximo salto, o encapsulamento dos pacotes, entre outros. Completada essa tarefa, os pacotes são enfileirados na respectiva interface de saída, onde entra em ação o controle de tráfego.

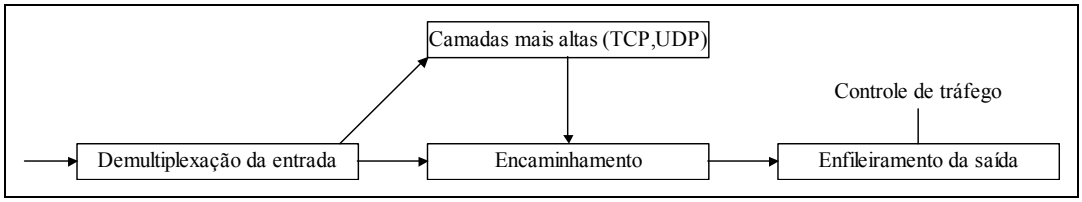

Figura 2.9 - Processamento dos dados de rede

O controle de tráfego do Linux[14] é composto pelos seguintes componentes conceituais: disciplinas de enfileiramento, classes e filtros de classificação e policiamento. Cada interface de rede tem associada sua disciplina de enfileiramento, que controla como é tratado o enfileiramento neste dispositivo.

[14] Um mecanismo semelhante está disponível para o sistema operacional Windows, através da interface Winsock2 GQoS (Bernet, 1998), porém esta API não oferece toda a flexibilidade nem a possibilidade de hierarquização do recurso como provê o LinuxTC.

Uma disciplina de enfileiramento pode ser simples tal qual aquela constituída apenas de uma única fila, como também pode ser mais elaborada e complexa, utilizando filtros para distinção dos pacotes, distribuindo-os entre classes (Figura 2.10).

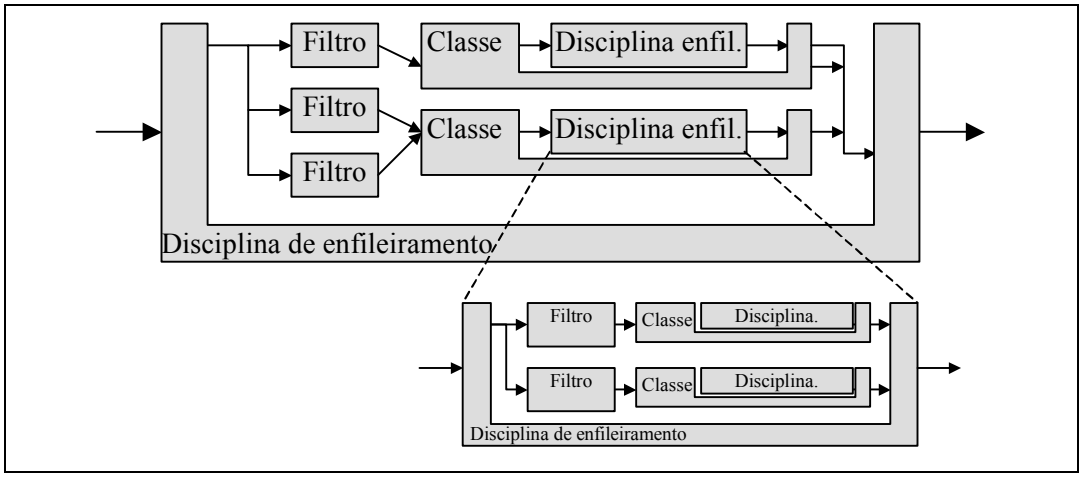

Figura 2.10 - Modelo de disciplina de enfileiramento, definido em (Almesberger, 1999)

O LinuxTC permite decidir a forma com que os pacotes devem ser enfileirados e quando eles devem ser descartados, numa situação de tráfego excedendo ao limite, por exemplo. É possível definir a ordem de envio desses pacotes, aplicando-se prioridades aos fluxos e, por último, retardar o envio de alguns pacotes para limitar a taxa de dados do tráfego de saída. Executadas todas essas operações, cada pacote pode ser entregue ao driver da interface para a transmissão pela rede.

Com este conjunto de ferramentas, é possível configurar vários tipos de políticas para o escalonamento dos pacotes, distribuídas em uma estrutura hierárquica. Cada classe de uma disciplina de enfileiramento pode possuir uma nova disciplina para a qual será delegado o escalonamento de pacotes pertencentes àquela classe. Essa estrutura hierárquica pode ser visualizada de uma forma mais clara através da Figura 2.11, onde as classes podem ser vistas como conectores entre as disciplinas de enfileiramento. O conceito de árvore de recursos virtuais, que será introduzido no Capítulo 3, é genérico o suficiente para abranger também a utilização do controle de tráfego do Linux em uma instanciação da arquitetura proposta por este trabalho.

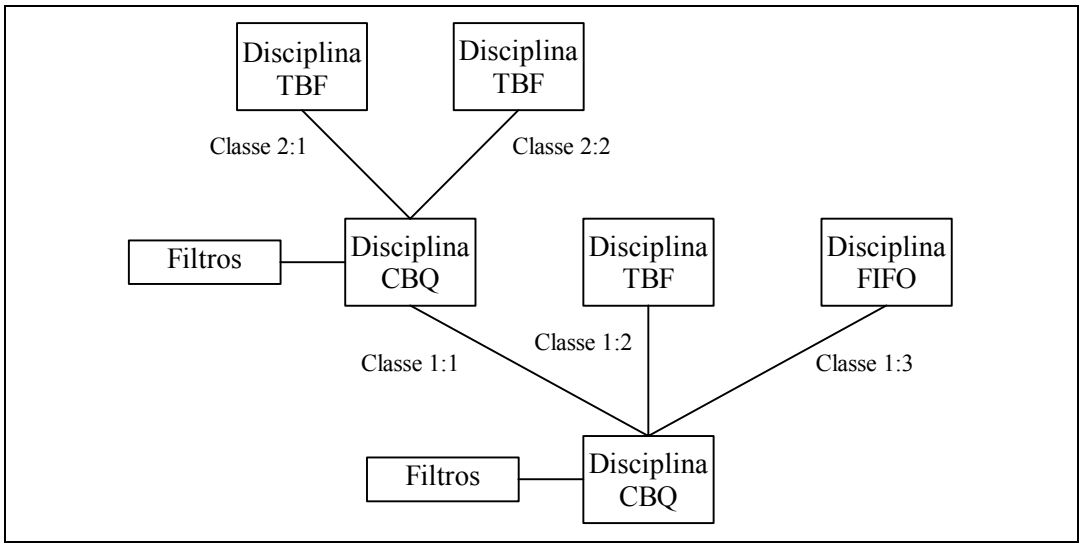

Figura 2.11 - Exemplo de hierarquia de disciplinas de enfileiramento, utilizando uma notação em árvore

2.6
Suporte à Adaptabilidade no Linux

Como já definido, o conceito de adaptabilidade está relacionado à capacidade que possui um provedor para a configuração e provisão de novos serviços de comunicação e processamento, através da inclusão ou modificação de estruturas como, por exemplo, as que definem novas políticas de provisão de qualidade de serviço, em tempo de execução.

Em sistemas operacionais, são encontradas duas formas básicas para a introdução de novos serviços: através do uso de algoritmos configuráveis, que permitem uma mudança de seu próprio comportamento a partir da modificação de seus parâmetros de operação; e através da inserção ou substituição de partes de código no conjunto de instruções que compõem o gerenciamento de recursos do sistema.

O algoritmo LDS, já apresentado, é um exemplo da primeira forma de adaptabilidade. A segunda forma é encontrada freqüentemente em sistemas baseados em *microkernel*, facilitada pela arquitetura do sistema, discutida anteriormente. Já em sistemas monolíticos, é observada a dificuldade de introdução de módulos ao *kernel* em tempo de execução.

O *kernel* monolítico do sistema operacional Linux (Maxwell, 2000), contudo, possui um subsistema de gerência de módulos de *kernel*, que inclui funções de inserção, remoção, verificação da necessidade e teste de utilização para os módulos. Este subsistema foi projetado inicialmente para a implementação de drivers de dispositivos e a conseqüente redução do tamanho do *kernel*. Vários trabalhos na área de sistemas operacionais utilizaram esta funcionalidade para realizar a configuração de partes internas do *kernel*, como o escalonamento de processos (Barabanov, 1997). Nota-se que, por não ser essa a finalidade prevista para esse subsistema e por não ser a adaptabilidade uma filosofia de projeto do Linux, várias são as modificações que devem ser feitas no próprio *kernel* para que ele aceite e utilize uma nova estratégia de escalonamento de processos, por exemplo.

De uma forma geral, o *kernel* do Linux deve ser modificado para tornar adaptáveis certos mecanismos cuja configuração dinâmica a partir de novos algoritmos é interessante. Abrir essas "brechas" no *kernel* traz a necessidade de políticas de segurança capazes de impedir que a introdução de um novo módulo prejudique o funcionamento do sistema. A própria estrutura programada para receber os novos algoritmos pode prover alguma proteção ao sistema, como ocorre com a estrutura hierárquica de escalonamento ao garantir o isolamento entre as categorias de aplicações. Mas muitas são as alternativas para que um usuário mal intencionado promova um congelamento do sistema ou quebre seu sistema de segurança. A discussão sobre esse assunto desvia-se dos objetivos principais do presente trabalho e, portanto, não será aqui aprofundada.

Exemplos de módulos interessantes para serem inseridos em um sistema em tempo de execução para a criação de um novo serviço de provisão de QoS são: estratégias de admissão de processos e de fluxos de rede; estratégias de escalonamento de processos e de pacotes; estratégias de monitoração da carga imposta a um certo recurso; estratégias de classificação de fluxos de pacotes, entre outros.

2.7 Resumo do Capítulo

O presente Capítulo apresentou uma discussão sobre os subsistemas de processamento e comunicação dos GPOS, ressaltando as deficiências que impedem a predição do seu comportamento mediante as diferentes situações de carga e necessidades das aplicações. Em seguida, foi proposto um modelo simplificado que abrange de forma genérica os recursos de processamento e comunicação envolvidos na execução de aplicações multimídia distribuídas. Com esse modelo, a dependência existente entre os fluxos de dados e de instruções foi evidenciada, o que levou à conclusão de que existe a necessidade de orquestração de recursos internamente ao sistema operacional, entre CPU e buffers de comunicação. A partir das diferentes formas de implementação da pilha de protocolos, o modelo mostrou que cada um dos processos (sejam de aplicações, sejam de entidades de protocolo) e cada uma das filas de comunicação devem ter garantidas suas necessidades de uso de tais recursos. Tornou-se imperativo, portanto, que essa abordagem fosse considerada na construção da arquitetura.

Vista a necessidade de provisão de QoS em sistemas operacionais, vários trabalhos sobre o assunto foram descritos. Além de expor as soluções propostas para cada um dos pontos críticos, o texto deste Capítulo procurou identificar as estruturas e mecanismos definidos em comum, entre as diversas implementações. Foram vistas alternativas para o escalonamento de recursos, entidades de controle de admissão, gerenciadores de recursos, estratégias de mapeamento, formas de adaptação de serviços em sistemas operacionais, métodos para a solicitação de serviços e estabelecimento de contratos, entre outros. Esse estudo forneceu o embasamento necessário para a definição dos elementos que compõem a arquitetura proposta no Capítulo 3.

3
Descrição da Arquitetura

Este Capítulo apresenta a proposta de uma arquitetura para a provisão de QoS nos subsistemas de comunicação e processamento de sistemas operacionais. A arquitetura QoSOS, como foi denominada, foi definida a partir da especialização dos frameworks genéricos descritos em (Gomes, 1999), acrescidos de algumas novas estruturas aqui introduzidas.

O processo de especialização dos frameworks compreende o preenchimento de pontos de flexibilização (*hot-spots*) (Pree, 1995). No contexto dos frameworks genéricos descritos em (Gomes, 1999), o preenchimento dos pontos de flexibilização ocorre em duas etapas distintas do ciclo de vida de um serviço (Colcher, 2000). Na fase de construção do serviço, os *hot-spots* em questão representam características particulares dos subsistemas envolvidos na provisão e, por isso, são chamados de *hot-spots* específicos do ambiente. Dessa forma, existe a possibilidade do reuso da mesma estrutura para a modelagem de um mecanismo adaptável, para diferentes partes do sistema. Na fase de operação, os chamados *hot-spots* específicos do serviço são preenchidos conforme a demanda por novos serviços. Essa funcionalidade da arquitetura garante a utilização de um mesmo ambiente para a provisão de diferentes serviços.

A Figura 3.1 mostra como os tipos de *hot-spots* descritos podem ser completados para a construção de uma arquitetura de provisão de QoS em sistemas operacionais. Os frameworks genéricos definem as estruturas para a provisão de QoS, as quais são comuns aos vários subsistemas que participam do fornecimento do serviço fim-a-fim. A primeira etapa de especialização é feita para que sejam incluídas funcionalidades específicas de sistemas operacionais, como os mecanismos pertinentes aos subsistemas de escalonamento de processos e de comunicação em rede. A etapa seguinte de particularização define aspectos relacionados à provisão do serviço, como o conjunto de políticas de QoS que cada um dos subsistemas disponibilizará a seus usuários.

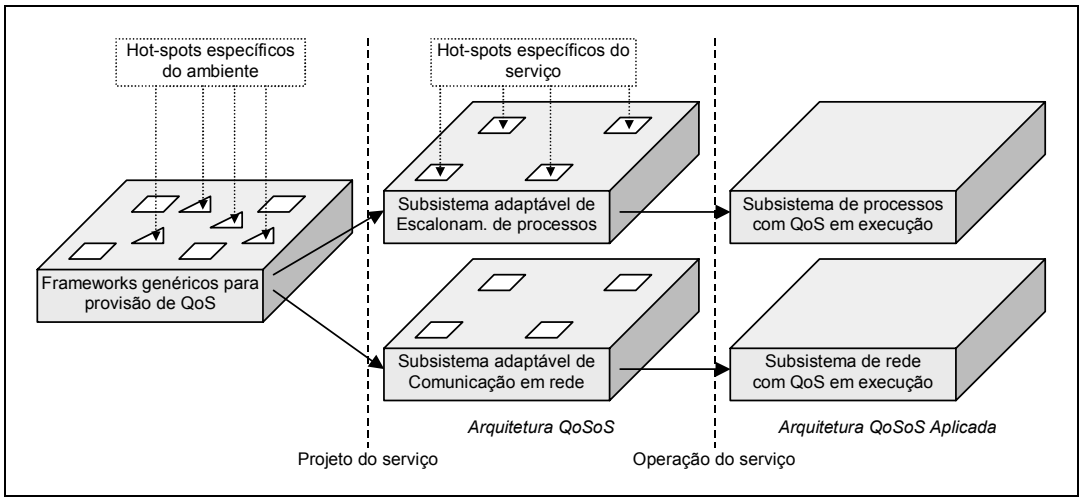

Figura 3.1 -Tipos de *hot-spots* de uma arquitetura modelada pelos frameworks genéricos de (Gomes, 1999)

A forma adotada neste Capítulo para a descrição da arquitetura QoSOS segue as etapas de preenchimento dos *hot-spots*, como mostrado pela figura anterior. A apresentação de cada um dos elementos da arquitetura, obtidos a partir da especialização dos frameworks genéricos, é acompanhada de um exemplo de sua aplicação, especificamente sobre o subsistema de escalonamento de processos. Exemplos de aplicação dos frameworks para o subsistema de comunicação em rede ficarão reservados para o Capítulo 4, já que este relata a implementação de um cenário de uso sobre tal subsistema.

As mesmas subdivisões propostas por (Gomes, 1999) foram utilizadas neste Capítulo para estruturar a descrição da arquitetura, com a adição de uma quarta subdivisão, como segue:

- Parametrização de Serviços;
- Compartilhamento de Recursos;
- Orquestração de Recursos;
- Adaptação de Serviços.

A descrição utiliza uma abordagem orientada a objetos, em notação que segue a Linguagem de Modelagem Unificada (UML, 1997). Adotou-se a convenção de que as classes-base da arquitetura apresentam-se preenchidas com a cor cinza. As classes que definem possíveis instanciações das classes-base estão coloridas de branco. Quando a hierarquia de derivação de uma classe-base não for

ilustrada, será utilizado um adorno do tipo "`<<classe-base>>`" sobre o nome da classe final, para indicar que se trata de uma especialização daquela classe-base. Finalmente, na descrição textual, as classes e métodos abstratos estão notados de forma distinta aos elementos concretos, em `itálico`.

3.1 Parametrização de Serviços

O framework para parametrização de serviços modela uma estrutura responsável por definir um esquema de parâmetros de caracterização de serviços, de forma genérica, independente dos possíveis serviços a serem oferecidos pelo sistema operacional. Tais parâmetros descrevem o comportamento tanto dos fluxos dos usuários quanto dos subsistemas que compõem o sistema operacional. Nota-se que ambas caracterizações partem da solicitação do serviço por parte do usuário, quando este informa quais os requisitos de processamento e comunicação desejados (*parâmetros de especificação de QoS*) e qual a dinâmica de geração dos seus dados (*parâmetros de caracterização de carga*). Um terceiro conjunto de parâmetros pode disponibilizar informações sobre o estado interno de um subsistema, como a quantidade de recursos disponíveis aos usuários (*parâmetros de desempenho do provedor*).

É importante observar, ainda, que cada nível de abstração e cada subsistema que compõe a infra-estrutura de provisão de serviços possui uma forma distinta de descrição dos parâmetros citados. Por exemplo, a especificação de QoS para o subsistema de escalonamento de processos pode ser descrita, como já visto, por parâmetros como percentagem de uso da CPU, número de instruções a serem executadas em um intervalo de tempo ou o par quantum e período de execução. Já o subsistema de rede de um sistema operacional pode oferecer parâmetros como largura de banda, retardo máximo e taxa de perda de pacotes. Cabe a mecanismos atuantes durante a negociação de QoS do sistema operacional o mapeamento dos parâmetros de um nível de abstração superior para os parâmetros que descrevem o comportamento dos recursos de tais subsistemas. A *hierarquia de parâmetros* oferecida pelo framework possibilita a criação de parâmetros abstratos que devem ser especializados conforme as circunstâncias particulares, promovendo a

generalidade necessária para a definição de parâmetros em diferentes níveis de abstração.

As *políticas de provisão de QoS*, distribuídas pelos elementos da arquitetura, determinam como os recursos serão escalonados, quais as condições para a admissão de novos fluxos, quais os métodos para a criação de recursos virtuais, entre outras funcionalidades. Para descrever o comportamento do sistema na provisão do serviço solicitado, muitas dessas políticas se baseiam em parâmetros de caracterização de serviços contidos em uma "base de informações de QoS" que se encontra distribuída pelo sistema. Por isso, a coexistência de diversos parâmetros para a solicitação de diferentes serviços deve ser organizada de forma a facilitar a aplicação de tais políticas.

As chamadas *categorias de serviço* agrupam conjuntos de parâmetros que possuem características em comum, relacionadas ao tipo de ambiente, nível de visão de QoS, tipos de dados e de necessidades do usuário. A possível associação das políticas às categorias de serviço simplifica, então, a ação dos mecanismos, já que a manipulação dos parâmetros ocorrerá no contexto da categoria desejada, em uma espécie de interface para a base de informações de QoS do sistema.

Por exemplo, numa estação em que o modelo IntServ (Braden, 1994) é utilizado para a provisão de QoS, as categorias de serviço disponibilizadas pelo sistema são serviço garantido e carga controlada. A categoria de serviço garantido utiliza os parâmetros RSpec (Shenker, 1997a), para a especificação da QoS, e TSpec (Shenker, 1997b), para caracterização do tráfego, oferecendo aos usuários um serviço com garantia de retardo máximo e sem perda de pacotes. A categoria de carga controlada utiliza apenas o parâmetro TSpec, já que não são dadas garantias específicas sobre a QoS, oferecendo apenas um serviço sem congestionamento no uso dos recursos. Dessa forma, o sistema operacional deve possuir mecanismos distintos para o tratamento dos fluxos submetidos sob tais categorias, como estratégias de escalonamento e de admissão.

No framework, as categorias de serviço também são estruturadas em uma hierarquia de derivação, o que permite que certas informações e operações, comuns a diferentes categorias, possam ser apresentadas em níveis superiores da

hierarquia, promovendo um reaproveitamento de código. A definição de categorias de serviço abstratas é muito útil no momento da definição de interfaces de solicitação de serviços que visam generalizar a invocação de métodos, independentemente dos mecanismos de provisão de QoS. Um exemplo dessa interface será mostrado no Capítulo 4.

3.1.1 Elementos do Framework para Parametrização de Serviços

A Figura 3.2 apresenta o framework para parametrização de serviços, especializado para a arquitetura QoSOS, onde a classe abstrata `ServiceCategory` simboliza a hierarquia de categorias de serviço. Os objetos dessa classe possuem, por associação, um ou mais parâmetros de caracterização de serviços, os quais são objetos da classe abstrata `Parameter`. O pattern estrutural *bridge* (Gamma, 1995) foi utilizado para representar as categorias de serviço como conjuntos de parâmetros, pelo relacionamento de agrupamento `parameterList`. Dessa forma, permite-se o desacoplamento entre as hierarquias de categorias de serviço e de derivação de parâmetros, tornando-as independentemente extensíveis.

O atributo `style` de `ServiceCategory` define o *estilo de compartilhamento*, considerado por (Mota, 2001) na aplicação dos frameworks para provisão de QoS na Internet. O conceito de estilo de compartilhamento permite que um mesmo conjunto de valores de parâmetros de serviço seja aplicado a fluxos distintos, mas que possuem alguma característica em comum. No caso de fluxos de pacotes, um exemplo de característica comum poderia ser o endereço do receptor. Para fluxos de instruções, poderia ser especificado que os mesmos valores de parâmetros seriam providos para um conjunto de threads de um mesmo processo. Esse conceito é originado dos estilos de reserva descritos no protocolo RSVP (Braden, 1997). Um estilo pode especificar, por exemplo, um serviço compartilhado (*shared*) ou um serviço exclusivo (*fixed*) para os fluxos especificados pelo usuário.

Para envolver as categorias de serviços dos subsistemas de escalonamento de processos e de enfileiramento de pacotes, a arquitetura QoSOS determina a especialização da classe `ServiceCategory` entre as classes abstratas `ProcessingServiceCategory` e `QueuingServiceCategory`, respectivamente.

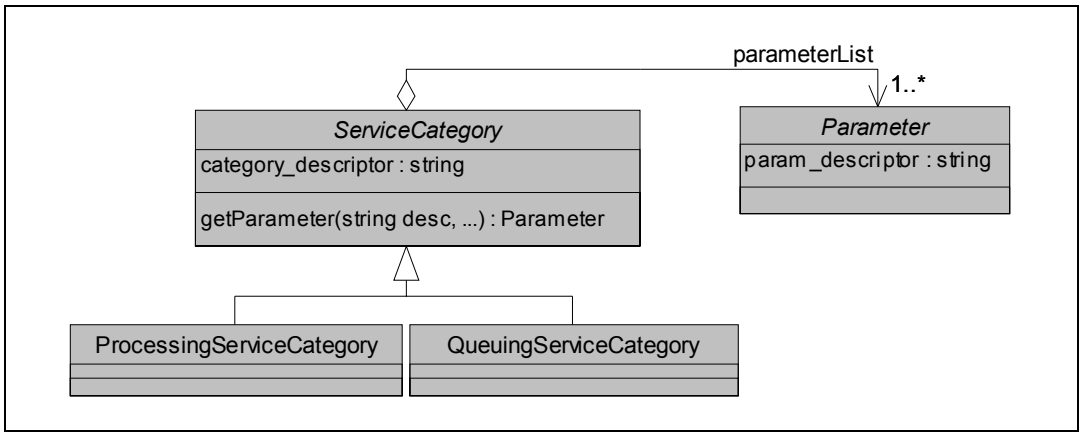

Figura 3.2 - Framework para Parametrização de Serviços

3.1.2 Exemplo de Aplicação do Framework para Parametrização de Serviços

Um exemplo de aplicação do framework de parametrização de serviços pode ser visto na Figura 3.3. O exemplo representa uma instanciação para o subsistema de escalonamento de processos, onde as categorias de serviço oferecidas correspondem às classes de aplicação definidas na Seção 2.2.2, que possuem necessidades especiais de provisão de QoS. A hierarquia de derivação de categorias de serviço mostra a classe `ProcessingServiceCategory` especializada pelas classes `SoftRealTimeServiceCategory` (para aplicações de tempo real suave) e `HardRealTimeServiceCategory` (para aplicações de tempo real severo).

A hierarquia de parâmetros de caracterização de serviços propõe a especialização da classe `Parameter` para definir os parâmetros peso (classe `Weight`) e o par quantum/período de execução (classe `PeriodicSpec`). O peso atribuído à execução de um processo corresponde, de maneira relativa, à percentagem de uso da CPU a ele reservada, o que, normalmente, é um requisito

de processamento de aplicações de tempo real suave. Portanto, um objeto da classe `SoftRealTimeServiceCategory` deve ter associado um objeto da classe `Weight`, através do método `addParameter`. De forma análoga, o parâmetro `PeriodicSpec` deve ser adicionado aos objetos da classe `HardRealTimeServiceCategory`, por ser o período e quantum de execução as características e requisitos de processamento desta categoria.

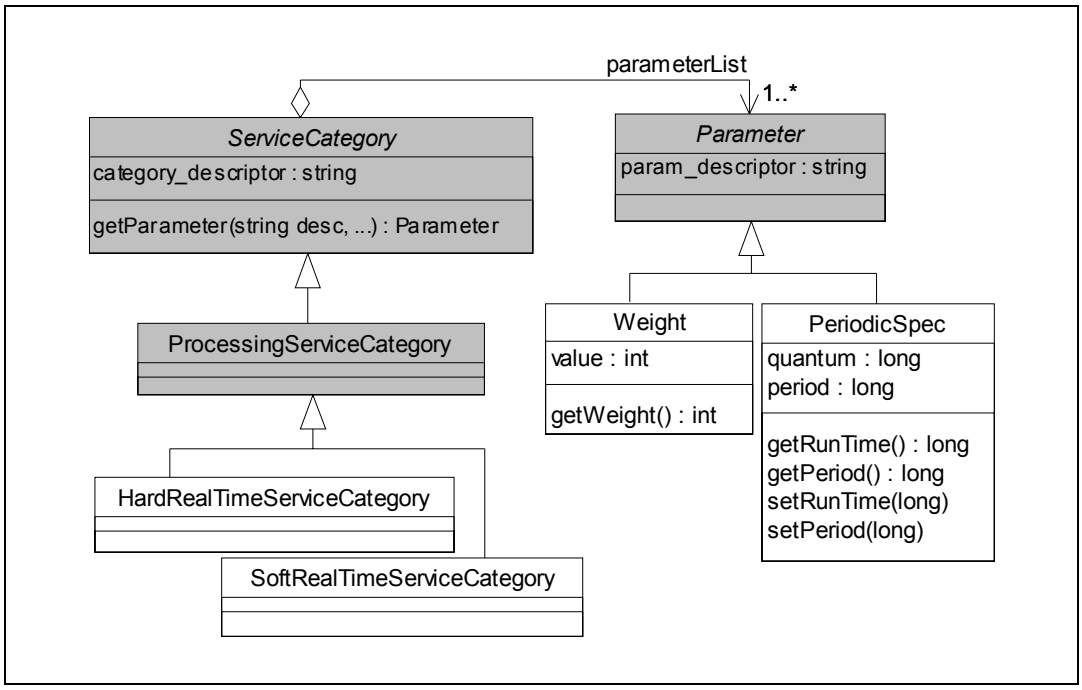

Figura 3.3 - Exemplo de aplicação do Framework para Parametrização de Serviços

3.2 Compartilhamento de Recursos

Os frameworks para compartilhamento de recursos se baseiam no conceito de *recurso virtual* para modelar os mecanismos de alocação e escalonamento de recursos. Recursos virtuais são parcelas de utilização de um ou mais recursos reais distribuídas entre os fluxos submetidos pelos usuários.

Para facilitar o emprego de vários algoritmos de escalonamento sobre um mesmo recurso e, assim, oferecer um conjunto amplo e flexível de serviços em um mesmo sistema, os recursos virtuais são dispostos em uma estrutura chamada *árvore de recursos virtuais*. Cada recurso real possui uma árvore de recursos virtuais associada, embora uma mesma árvore de recursos possa representar a

estrutura de escalonamento sobre mais de um recurso real. Um exemplo de árvore sobre vários recursos pode ser dado para representar a estrutura de escalonamento de processos em sistemas multiprocessados.

A Figura 3.4 apresenta um exemplo de árvore de recursos virtuais para o recurso real CPU. As folhas representam os recursos virtuais, no caso, parcelas de utilização da CPU para cada thread alvo do serviço. A raiz da árvore corresponde ao escalonador de mais baixo nível da hierarquia, aquele que realmente distribui o tempo de uso do recurso entre os nós filhos. Adicionalmente, este escalonador, denominado *escalonador de recurso raiz*, pode permitir que os recursos virtuais filhos utilizem diferentes recursos reais de um mesmo tipo, paralelamente. Os nós intermediários da árvore são recursos virtuais especializados, responsáveis por ceder a sua parcela de utilização do recurso real aos seus recursos virtuais filhos. Denominados *escalonadores de recursos virtuais*, esses nós podem ter acesso a mais de um recurso real por vez, para também permitir que seus nós filhos sejam atendidos de forma paralela.

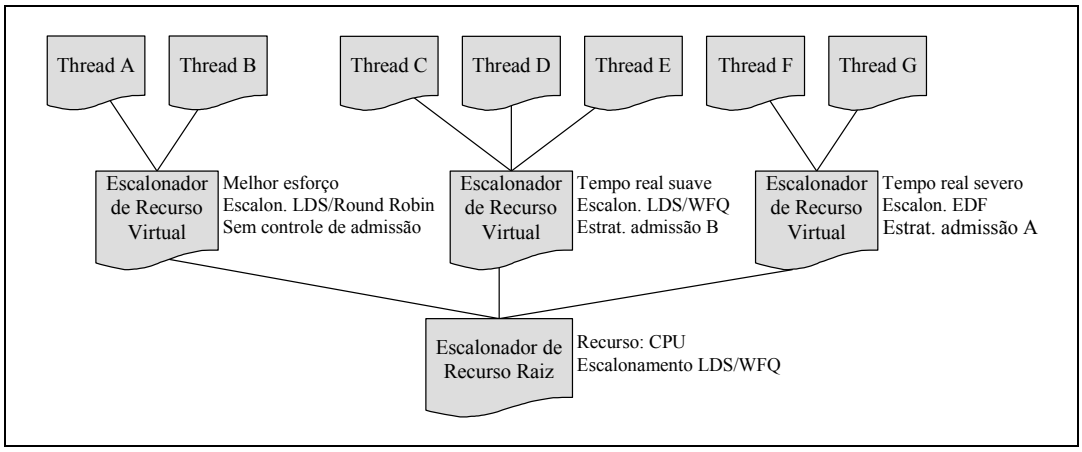

Figura 3.4 - Exemplo de árvore de recursos virtuais para o recurso CPU

Cada escalonador de recurso virtual está associado a uma categoria de serviço e às políticas de provisão de QoS correspondentes, como as *estratégias de escalonamento* e *de admissão*, além de um *componente de criação de recursos virtuais*. Para efetivar a criação de um recurso virtual, esse componente executa tarefas como a adição do recurso virtual à lista de responsabilidades do escalonador e a configuração dos módulos de classificação e de policiamento.

3.2.1
Elementos do Framework para Escalonamento de Recursos

A Figura 3.5 ilustra o framework para escalonamento de recursos, como definido em (Gomes, 1999). Na arquitetura QoSOS, ele deve ser especializado para os recursos CPU e buffers de comunicação, como será apresentado nos exemplos. As classes `VirtualResource`, `RootResourceScheduler` e `VirtualResourceScheduler` representam, respectivamente, os recursos virtuais, os escalonadores de recursos reais e os escalonadores de recursos virtuais. A classe abstrata `ResourceScheduler` uniformiza o acesso a esses escalonadores, através da definição do método `schedule()`. Esse método permite que os escalonadores desativem o recurso virtual filho detentor do direito de uso do recurso real (método `deactivate()`) e ativem o recurso virtual filho seguinte a obter esse direito (método `activate()`).

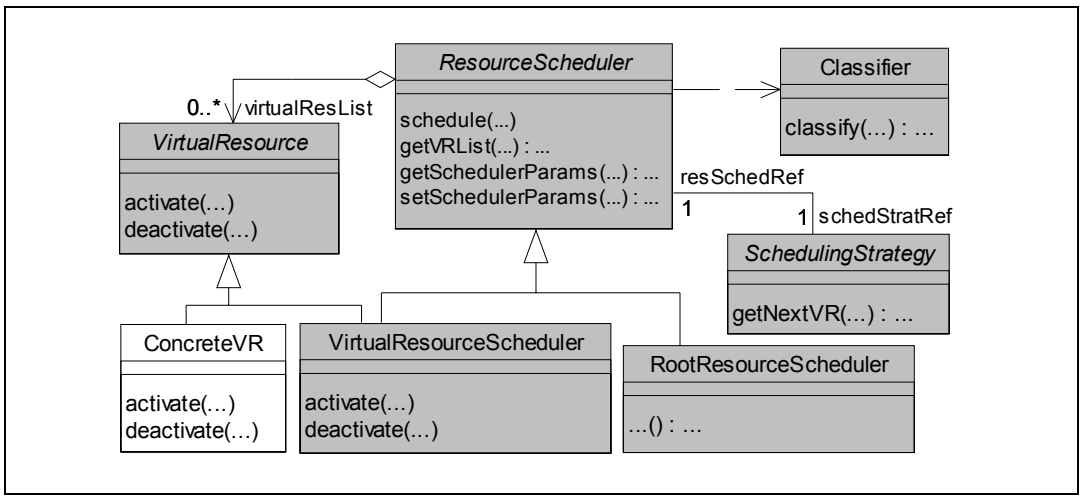

Figura 3.5 - Framework para Escalonamento de Recursos

Para que seja atribuída aos objetos da classe `VirtualResourceScheduler` a capacidade de obter parcelas de utilização do recurso (agindo como um recurso virtual) e de redistribuí-las entre seus filhos (agindo como um escalonador), essa classe possui relacionamentos de especialização com `VirtualResource` e `ResourceScheduler`. Esse conjunto de relacionamentos define a estrutura da árvore de recursos virtuais: na raiz existe uma instância da classe `RootResourceScheduler`, nos nós intermediários instâncias da classe `VirtualResourceScheduler` e, por fim, nas folhas há instâncias da classe `ConcreteVR`.

A escolha do próximo recurso virtual a ser escalonado é realizada pela instância da classe `SchedulingStrategy` relacionada ao escalonador, uma vez que corresponde à estratégia de escalonamento definida para a categoria de serviço solicitada. Nesse caso, o uso do pattern comportamental *Strategy* (Gamma, 1995) determina um *hot-spot* específico do serviço, pois a flexibilidade no relacionamento entre o escalonador e a estratégia de escalonamento em uso permite a substituição de estratégias sem necessidade de alteração no mecanismo de escalonamento em si.

A classe `Classifier` representa o classificador que, por meio do método `classify()`, obtém a categoria de serviço da unidade de informação a ser escalonada, com o objetivo de direcionar essa unidade para o escalonador correspondente.

3.2.2 Exemplo de Aplicação do Framework para Escalonamento de Recursos

A Figura 3.6 mostra um exemplo de aplicação do framework de escalonamento de recursos que reflete a estrutura definida pela árvore de recursos virtuais da Figura 3.4, sobre o sistema Linux. A utilização do algoritmo LDS como escalonador na raiz da hierarquia e de algoritmos mais específicos nos escalonadores folhas foi proposta na Seção 2.4.3. O escalonador raiz (classe `RootCPUScheduler`), que deve estar implementado no *kernel* do sistema, tem a responsabilidade de ceder as parcelas de utilização da CPU a outros escalonadores (classe `VirtualCPUScheduler`), também implementados no *kernel*, os quais atribuem o direito de utilização às threads (classe Thread).

O sistema operacional Linux implementa threads no nível do *kernel* (Walton, 1996), determinando a existência de um único escalonador tanto para processos quanto para threads. Assim, é permitido o paralelismo entre threads de um mesmo processo e não existe o bloqueio de várias threads provocado por uma chamada de sistema. Fica facilitada a aplicação das prioridades de execução entre todas as threads existentes, se essa forma de escalonamento for desejada. Na realidade, uma thread no sistema Linux pode ser vista como um novo processo

que pode compartilhar alguns elementos com o processo pai, como área de memória e arquivos abertos. Por isso, no exemplo aqui apresentado, o nome da classe `Thread` foi utilizado para generalizar qualquer entidade de processamento.

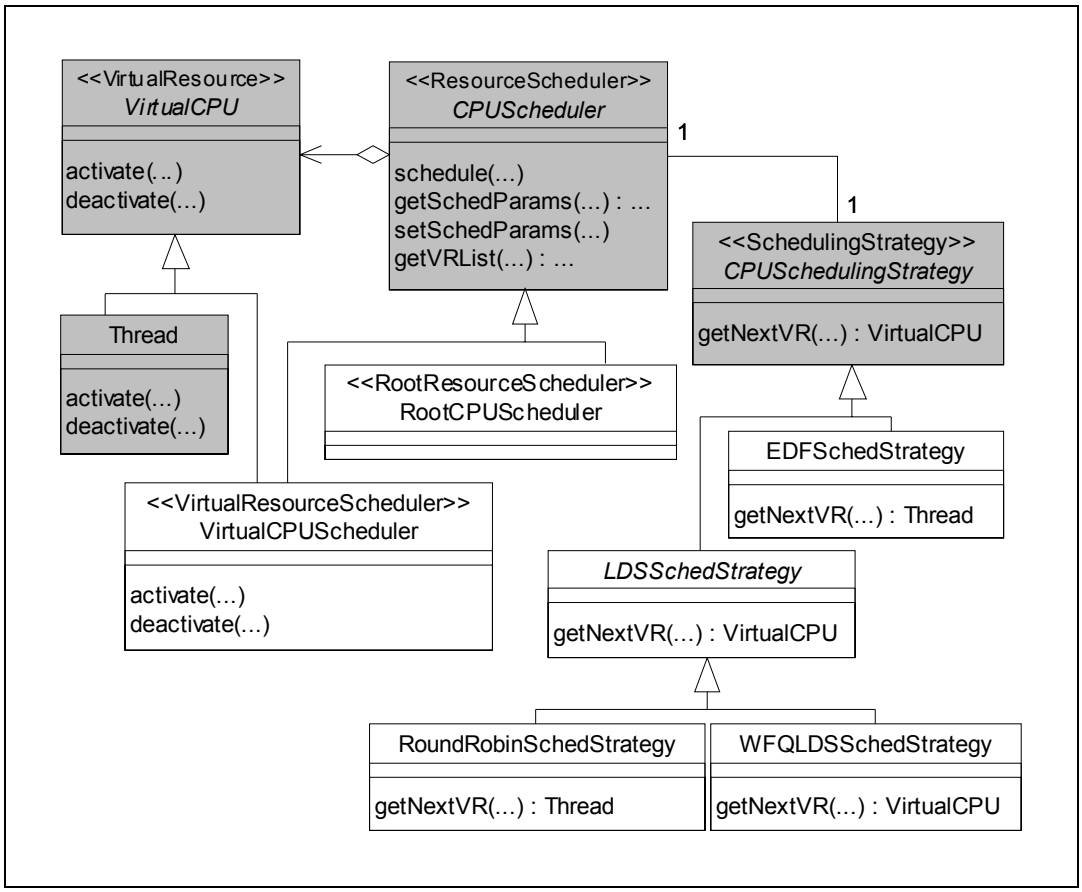

Figura 3.6 - Exemplo de aplicação do Framework para Escalonamento de Recursos

As estratégias de escalonamento disponibilizadas compreendem os seguintes algoritmos: LDS emulando o algoritmo WFQ para o escalonador de recurso raiz e a categoria tempo real suave; algoritmo EDF para a categoria de tempo real severo; e LDS emulando o algoritmo Round Robin para a categoria best-effort. Essas estratégias correspondem, respectivamente, às classes `WFQLDSStrategy`, `EDFSchedStrategy` e `RRobinLDSStrategy`. Nota-se que não só a definição de estratégias de escalonamento representa uma flexibilidade da arquitetura QoSOS, como também a utilização do algoritmo LDS compõe um outro *hot-spot* específico do serviço, em tempo de operação, que pode ser utilizado pela interface de adaptação de serviços.

Em estações com multiprocessamento simétrico (SMP), tanto os escalonadores de recursos virtuais quanto o escalonador de recurso raiz podem

adotar várias estratégias de divisão das parcelas de tempo das CPUs. Os escalonadores podem arbitrar se uma CPU estará dedicada exclusivamente a alguma categoria de serviço ou se será compartilhada proporcionalmente entre várias categorias.

Por exemplo, o escalonador original do sistema Linux define uma estratégia muito simples de elegibilidade de tarefas para execução em uma certa CPU (que no *kernel* é denominada cálculo do valor de excelência - função `goodness()`). A estratégia define pesos para os fatores prioridade e tempo já utilizado da CPU, somando bônus para as threads que pertencem ao processo em execução e para as threads que já vinham utilizando o mesmo processador em questão anteriormente.

3.2.3 Elementos do Framework para Alocação de Recursos

A Figura 3.7 apresenta o framework para alocação de recursos especializado para a arquitetura QoSOS. A classe abstrata `VirtualResourceFactory` representa os componentes responsáveis pela criação de recursos virtuais. O pattern de criação *Factory Method* (Gamma, 1995) modela o relacionamento de dependência entre esses componentes e os tipos de recursos virtuais por eles criados. Através das redefinições do método abstrato `createVR()`, as subclasses de `VirtualResourceFactory` implementam a instanciação das subclasses de `VirtualResource`. Cada escalonador presente em uma árvore de recursos virtuais está ligado a um componente de criação do respectivo tipo de recurso virtual (relacionamento entre as classes `ResourceScheduler` e `VirtualResourceFactory`).

A ligação entre componentes de criação e escalonadores permite que o método `createVR()` não somente crie o recurso virtual, como também adicione o recurso na lista de recursos virtuais de responsabilidade do escalonador. Adicionalmente, esse método dispara a configuração de classificadores (classe `Classifier`) que, através de uma lista de filtros (classe `Filter`), são capazes de identificar a categoria de serviço de cada unidade de informação dos fluxos. A utilização de classificadores é necessária, por exemplo, para identificar a categoria

de serviço de cada pacote transmitido em rede, tomando como base informações presentes no seu cabeçalho.

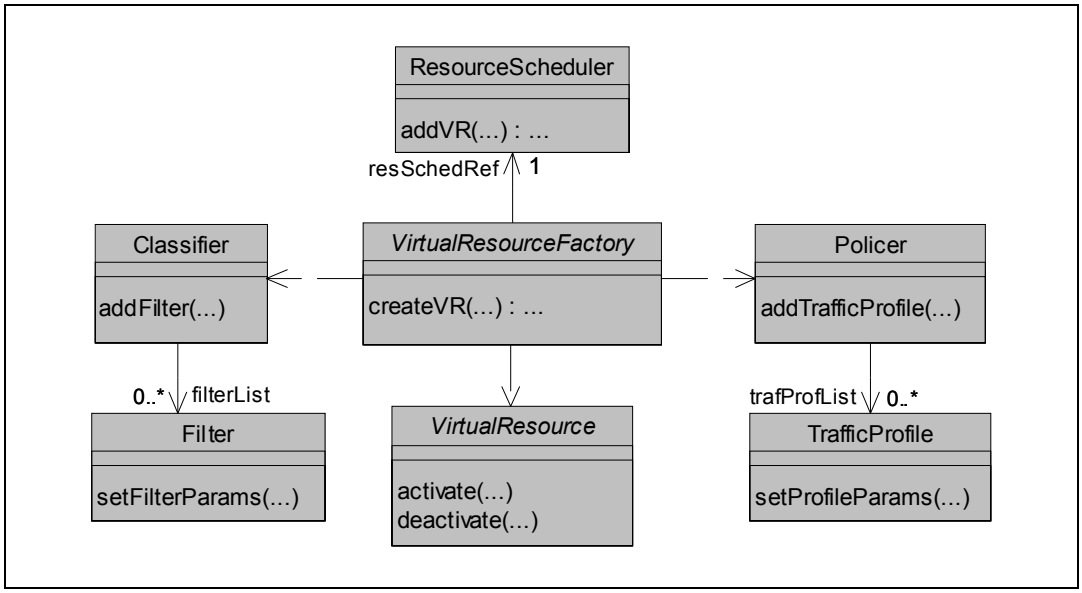

Figura 3.7 - Framework para Alocação de Recursos

Na criação de um recurso virtual deve-se, ainda, configurar o agente de policiamento (classe `Policer`) adicionando-se um perfil de tráfego (classe `TrafficProfile`) para caracterizar o fluxo que utilizará o recurso virtual. O agente de policiamento é responsável por verificar a conformidade do fluxo gerado pelo usuário em relação à caracterização por ele fornecida na solicitação do serviço. Os classificadores e agentes de policiamento foram acrescentados por (Mota, 2001) ao conjunto de frameworks genéricos para provisão de QoS.

3.2.4 Exemplo de Aplicação do Framework para Alocação de Recursos

A Figura 3.8 ilustra uma aplicação do framework para alocação de recursos, dando seqüência à estrutura de provisão dos exemplos anteriores. A classe `ThreadFactory` representa os métodos de criação que são comuns a qualquer tipo de thread definido pelas categorias de serviço. Tais tipos de threads são representados pelas especializações da classe `Thread`, que devem redefinir os métodos `activate()` e `deactivate()` conforme o comportamento exigido de cada um deles.

Por exemplo, threads de tempo real severo (classe `HardRTThread`) são caracterizadas pela execução repetitiva de seu código entre períodos máximos de tempo e durante um tempo determinado. O método de ativação dessas threads deve permitir essa seqüência de repetições, enquanto a desativação deverá ocorrer automaticamente, dentro do intervalo de tempo especificado.

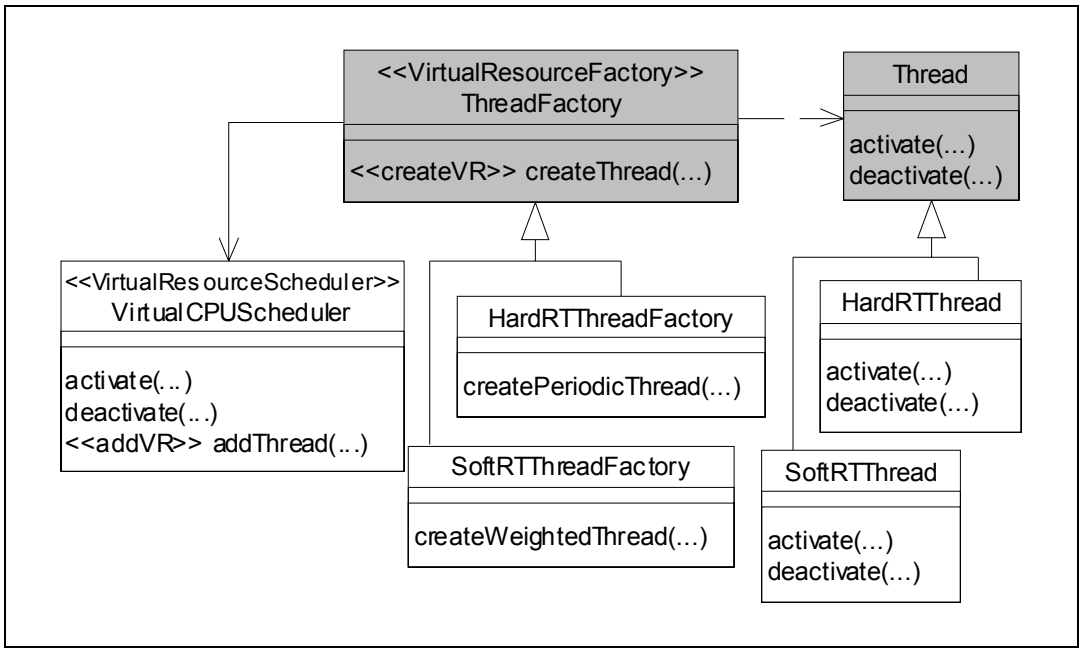

Figura 3.8 - Exemplo de aplicação do Framework para Alocação de Recursos

As especializações da classe `ThreadFactory` devem redefinir o método `createThread()` para que as características específicas de cada tipo de thread sejam consideradas no momento de sua criação. Threads de tempo real suave devem ser escalonadas seguindo estratégias de escalonamento que considerem o percentual de CPU especificado, enquanto threads de tempo real severo, como já visto, devem ser tratadas por escalonadores cujas estratégias levem em conta o período e tempo de execução. A classe `ThreadCreator`, por sua vez, utiliza o método `addThread()` de `VirtualCPUScheduler` para adicionar a thread criada ao escalonador correspondente.

Percebe-se que não houve qualquer instanciação de filtros de classificação nem de perfis de policiamento, devido à natureza ativa do recurso CPU. Nesse caso, não há necessidade de policiamento porque o próprio recurso *busca* as unidades de informação (instruções) conforme seu processamento, atendendo a um fluxo (thread) por vez, conforme determinado pelos escalonadores. Cada fluxo

possui, desde o momento de sua criação, todas as suas unidades de informação já armazenadas em algum tipo de memória. A referência a cada fluxo pode, então, ser mantida constantemente na fila de execução correspondente à categoria de serviço requisitada, o que elimina a necessidade de classificação.

Se estivessem sendo tratados os buffers de comunicação (como será visto no Capítulo 4), as ações de classificação e policiamento seriam necessárias devido à natureza passiva desses recursos. As filas de espera *recebem* os pacotes de dados para armazenamento, aguardando pela comunicação para o próximo nível. A freqüência e a assincronia com que essas unidades de informação chegam ao recurso demandam um controle de uso do espaço de armazenamento ou da largura de banda, que é realizado através do policiamento. Também em decorrência da assincronia de chegada, cada pacote deve ser analisado para que seja determinada a categoria de serviço do fluxo a que ele pertence e, conseqüentemente, seja identificada a fila na qual deve ser armazenado.

3.3 Orquestração de Recursos

Na área de atuação dos sistemas operacionais, vários são os recursos que devem ter seus mecanismos de escalonamento e de alocação gerenciados de forma integrada, de modo a viabilizar a orquestração dos recursos no ambiente como um todo. Na arquitetura QoSOS, a modelagem da orquestração de recursos é apresentada pela especialização de dois frameworks distintos: o framework para negociação de QoS e o framework para sintonização de QoS.

O framework para negociação de QoS modela os mecanismos de negociação e mapeamento que operam durante as fases de solicitação e estabelecimento de contratos de serviços, além dos mecanismos de admissão que atuam somente na fase de estabelecimento. Já o framework para sintonização de QoS modela os mecanismos de sintonização e monitoração que atuam na fase de manutenção do serviço. As *políticas de orquestração de recursos* da arquitetura QoSOS estão diretamente relacionadas à dependência existente entre os subsistemas de rede e de escalonamento de processos evidenciada no modelo proposto na Seção 2.3.

Ao receber um pedido de admissão de um serviço, o controlador de admissão do sistema operacional envia uma requisição a um *agente de negociação*, ou negociador. Este identifica todos os recursos reais que podem estar envolvidos no fornecimento do serviço e distribui entre eles a parcela de responsabilidade sobre a provisão da QoS especificada. No caso de uma aplicação distribuída, o negociador do sistema operacional identificará que as CPUs e os buffers de comunicação são os recursos que devem participar do oferecimento do serviço. A negociação de QoS em um sistema operacional é feita de forma centralizada, já que um único agente pode ter o conhecimento sobre todos os recursos do ambiente.

As aplicações multimídia distribuídas devem ter garantidas as suas necessidades sobre cada uma das threads que compõem seus processos, bem como sobre as threads que executam a pilha de protocolos (conforme evidenciado pelo modelo da Seção 2.3). Além disso, os buffers de comunicação compartilhados devem ser capazes de encaminhar os pacotes de acordo com a parcela de QoS atribuída à estação pelo protocolo de negociação de rede e, por isso, a forma de implementação do subsistema de rede deve ser considerada para a distribuição das responsabilidades. As *estratégias de negociação* definem como será a política de orquestração, baseando-se na implementação de subsistemas específicos.

Por exemplo, se a pilha de protocolos fosse implementada de modo que uma thread fosse dedicada a cada fluxo, como em (Mehra, 1996), as threads da aplicação poderia ter suas necessidades de processamento individualmente negociadas. Já a thread executando o protocolo e a fila compartilhada de envio poderiam dividir os requisitos estipulados pelo protocolo de negociação de rede.

Para um GPOS[15] onde a aplicação e a pilha de protocolos são processadas em um mesmo contexto de execução, a estratégia de negociação pode distribuir as responsabilidades da seguinte forma – baseado em (Lakshman, 1997) e (Campbell, 1996):

---

[15] Considera-se aqui que o GPOS já foi estendido pela arquitetura LRP, para que o recebimento de pacotes proceda de forma semelhante ao envio, ambos na prioridade de execução do processo responsável, no contexto de uma chamada de sistema.

- Os requisitos de desempenho a serem atribuídos à thread da aplicação equivalem à soma entre os requisitos necessários para a própria thread manipular os dados e para a pilha de protocolos processar os pacotes, em conformidade com a caracterização especificada. Caso seja difícil a identificação de algum parâmetro de desempenho, o negociador pode solicitar à aplicação uma comunicação-teste, com o propósito de obtê-lo através de medições.
- Os requisitos de desempenho a serem atribuídos à fila de pacotes de saída equivalem aos requisitos determinados pelo protocolo de negociação de rede para a estação.

A partir das parcelas de responsabilidade atribuídas a cada recurso, são acionados os mecanismos de *mapeamento*, consistindo na tradução da categoria de serviço (e dos valores dos parâmetros associados) especificada na solicitação do serviço para as categorias de serviço (e valores de parâmetros associados) específicas relacionadas diretamente com a capacidade de operação dos recursos reais envolvidos. O mecanismo de *controle de admissão de cada recurso* deve, então, ser acionado a fim de verificar a viabilidade de aceitação do novo fluxo, utilizando-se das *estratégias de admissão* de recursos virtuais em cada um dos escalonadores escolhidos de acordo com a categoria de serviço desejada. Se todos os controladores de admissão responderem de forma afirmativa, os mecanismos de *criação de recursos virtuais* são acionados. Caso contrário, a requisição pode ser imediatamente negada ou o mecanismo de negociação pode reiniciar o processo redistribuindo as parcelas de responsabilidade.

Durante a fase de manutenção de um contrato de serviço, ajustes sobre o sistema podem ser necessários para que sejam asseguradas as especificações de QoS já requisitadas pelos usuários. A monitoração dos recursos reais visa a identificação de qualquer disfunção operacional, seja por parte do usuário (e.g. fluxos submetidos fora da caracterização do tráfego), seja por parte do sistema (e.g. uma falha nos recursos, erros no cálculo das reservas). Os monitores devem, assim, emitir alertas ao mecanismo de sintonização na presença de algum distúrbio. As ações de sintonização podem envolver desde pequenos ajustes de

parâmetros em determinados escalonadores até a solicitação de uma renegociação de QoS para certos contratos de serviços.

3.3.1 Elementos do Framework para Negociação de QoS

A Figura 3.9 apresenta o framework para negociação de QoS especializado para a arquitetura QoSOS. A classe `QoSOSAdmissionController` representa o controlador de admissão do sistema operacional, responsável por receber os pedidos de admissão de serviços, por meio do método `admit()`. A classe `QoSOSNegotiator` implementa o agente de negociação de QoS no sistema operacional. A solicitação do serviço deve ser encaminhada pelo controlador de admissão a um objeto dessa classe através do método `request()`. Para atender à requisição, a classe `QoSOSNegotiator` solicita que um objeto da classe `NegotiationStrategy` calcule a parcela de responsabilidade de cada recurso na negociação (método `calcResponsibilities()`). Tais estratégias podem variar de acordo com a forma de implementação de certos subsistemas, como já visto, ou conforme outras restrições que podem ser aplicadas à orquestração de recursos. O pattern *Strategy* foi utilizado para permitir que a arquitetura possa adotar diferentes regras de divisão de responsabilidades. Assim, esse *hot-spot* específico do serviço confere à arquitetura QoSOS o suporte a diferentes cenários de orquestração.

Durante o processo de divisão, o negociador deve recorrer a uma instância da classe `QoSOSMapper` para promover a tradução (método `translate()`) dos parâmetros especificados na requisição do serviço para parâmetros específicos do sistema operacional, diretamente relacionados com os recursos envolvidos. Os mapeadores de QoS utilizam o método `map()`, provido pelas estratégias de mapeamento (classe `MappingStrategy`) que definem como devem ser feitas as traduções. Neste ponto, também foi utilizado o pattern Strategy, que habilita a arquitetura a manipular parâmetros de diferentes categorias de serviço.

As subclasses de `ResourceAdmissionController` (`ProcessingAdmController` e `QueuingAdmController`) devem ser

especializadas para que cada árvore de recursos virtuais tenha associado um controlador de admissão. Assim, o negociador QoSOS, de posse dos parâmetros específicos que devem ser garantidos para cada recurso envolvido, solicita o teste de viabilidade da reserva aos controladores de admissão correspondentes, através do método `admit()`. O pattern Strategy foi utilizado novamente, para permitir que os controladores de admissão utilizem estratégias (classe `AdmissionStrategy`) de acordo com a categoria de serviço solicitada.

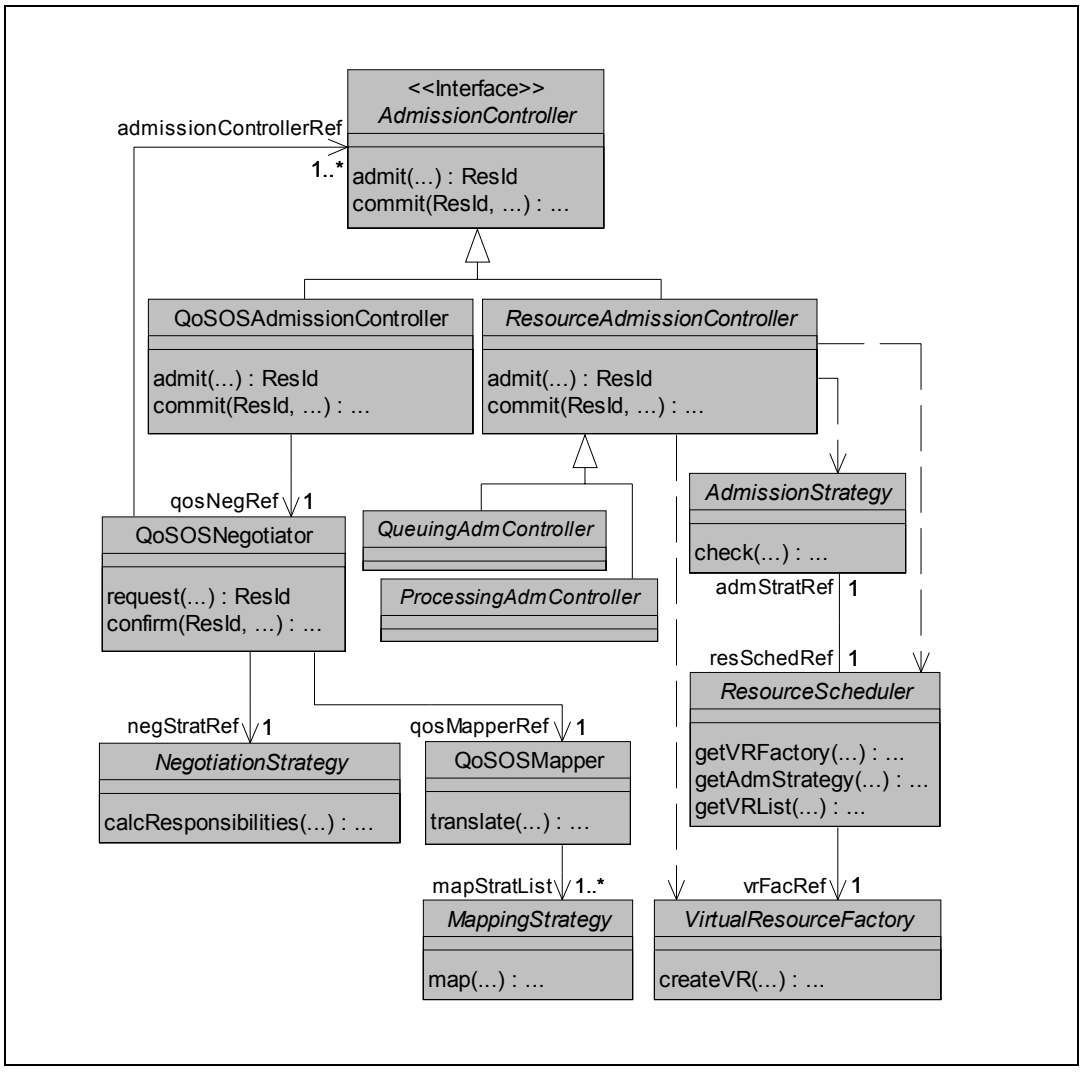

Figura 3.9 - Framework para Negociação de QoS

Se a estratégia de admissão retorna uma resposta afirmativa sobre a possibilidade de reserva (através do método `check()`), o controlador de admissão do recurso gera um identificador (`ResId`) para uma pré-reserva feita por ele. Essa pré-reserva não implica na criação do recurso virtual: as informações sobre alocação são utilizadas somente para que novas admissões considerem a sua

existência. Caso a reserva não seja confirmada dentro de um intervalo de tempo, essas informações deixam de ser consideradas. O negociador aguarda que todos os controladores de admissão retornem seus identificadores de reserva, para que ele possa gerar o seu próprio identificador e retorná-lo ao controlador de admissão do sistema operacional (classe `QoSOSAdmissionController`). Se algum dos controladores de admissão responder negativamente à requisição, o negociador não gera o seu identificador e retorna a impossibilidade de reserva dos recursos daquela solicitação. O controlador de admissão do sistema operacional, finalmente, responde ao solicitante sobre a viabilidade ou não da reserva dos recursos necessários ao pedido, pelo retorno do identificador gerado pelo negociador.

Para confirmar a solicitação do serviço, o método `commit()` de `QoSOSAdmissionController` é invocado pelo solicitante, passando como parâmetro o identificador de reserva fornecido anteriormente. Em seguida, o negociador é acionado pelo método `confirm()` para mapear o identificador recebido para os identificadores correspondentes gerados pelos controladores de admissão dos recursos. A confirmação é repassada a cada um dos controladores pelo método `commit()`, que consiste na reserva efetiva através do acionamento do componente de criação do recurso virtual (classe `VirtualResourceFactory`).

3.3.2 Exemplo de Aplicação do Framework para Negociação de QoS

A Figura 3.10 ilustra a aplicação do framework de negociação de QoS que segue o exemplo sobre o subsistema de escalonamento de processos. O controlador de admissão do sistema operacional e o agente de negociação foram instanciados pelas classes `QoSOSAdmissionController` e `QoSOSNegotiator`, respectivamente. A estratégia de negociação implementada pela classe `ProtocolOnSysCallStrategy` indica quais entidades de processamento devem ter requisitos de QoS negociados para que o desempenho do fluxo de dados seja garantido conforme a caracterização especificada.

Já que no Linux a pilha de protocolos é implementada no *kernel* e processada no contexto de uma chamada de sistema, os parâmetros de QoS a serem garantidos à thread da aplicação devem incluir as necessidades de processamento tanto para a aplicação tratar as informações quanto para o protocolo manipular os pacotes. Presume-se que o *kernel* do Linux, que serve como infra-estrutura para os exemplos, esteja estendido pela arquitetura LRP apresentada na Seção 2.5.5, para que todo o processamento de recebimento de pacotes seja executado efetivamente no contexto de uma chamada de sistema.

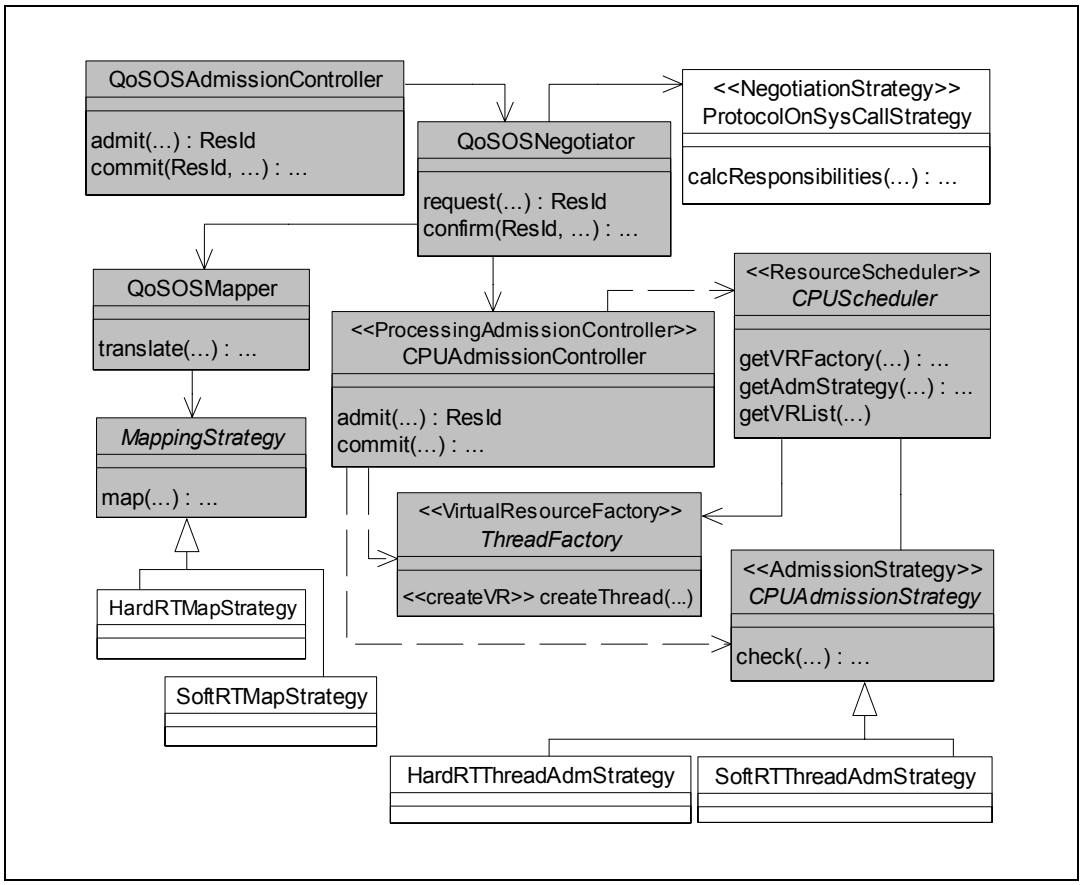

Figura 3.10 - Exemplo de aplicação do Framework para Negociação de QoS

A classe `QoSOSMapper` implementa o mecanismo de mapeamento acionado durante a divisão de responsabilidades e encarregado de fornecer os parâmetros específicos dos escalonadores da árvore de recursos da CPU. Para que esse mapeamento siga as necessidades exigidas pelas categorias de serviço, `QoSOSMapper` utiliza as estratégias implementadas nas classes `HardRTMapStrategy` e `SoftRTMapStrategy`, correspondentes às categorias `HardRealTimeServiceCategory` e `SoftRealTimeServiceCategory`, respectivamente.

Por exemplo, `HardRTMapStrategy` calcula o parâmetro período a partir de parâmetros que representam a taxa de informações, como a taxa de quadros de vídeo. Se um vídeo é gerado a uma taxa constante de 30 quadros por segundo, a thread da aplicação geradora deve ser executada para gerar um quadro a cada intervalo de 33 ms. A estratégia recorre à técnica de medição descrita na Seção anterior para calcular o quantum da aplicação.

Já a estratégia `SoftRTMapStrategy` calcula o peso que a thread deve receber seguindo também uma medição, desta vez para verificar o tempo de CPU necessário para que a aplicação realize a manipulação de dados durante um certo espaço de tempo. Devido à característica de processamento variável das aplicações de tempo real suave, essa medição é uma mera estimativa, mas, a partir dela, pode ser descoberta a percentagem de CPU necessária e, conseqüentemente, o peso que deve ser associado à thread no escalonamento. É importante ressaltar que estimativas mal formadas sobre o comportamento das entidades de processamento podem refletir em uma degradação da QoS, que pode ser identificada na fase de manutenção de contratos de serviço. Para isso, os mecanismos de sintonização de QoS agem na correção dessas anomalias, de forma que outros serviços em operação sejam pouco afetados.

A classe `CPUAdmissionController` implementa o mecanismo de controle de admissão sobre o recurso CPU, enquanto as estratégias definidas para cada categoria de serviço são implementadas pelas especializações da classe `CPUAdmissionStrategy`. Para aplicações de tempo real severo, a estratégia de admissão (classe `HardRTThreadAdmStrategy`, corresponde à estratégia A da Figura 3.4) verifica se a aceitação do novo recurso virtual não fará com que a porção de utilização da CPU alocada para essa categoria seja excedida. Essa restrição pode ser apresentada pela seguinte fórmula (Liu, 1973):

$$\sum_{i=1}^{N} \frac{quantum_i}{período_i} \leq P_{HRT},\ 0 \leq P_{HRT} \leq 1$$

onde N é o número de threads de tempo real severo já admitidas e P_{HRT} é a porção de CPU alocada à categoria. Adicionalmente, pode-se fazer um segundo teste para garantir que o quantum de cada thread seja completado antes que o

limite de tolerância dado pelo usuário (jitter) seja atingido. A fórmula é a seguinte (Campbell, 1996):

$$\sum_{i=1}^{N} \frac{quantum_i}{quantum_i + jitter_i} \leq 1$$

Para a estratégia de admissão de threads de tempo real suave (classe `SoftRTThreadAdmStrategy`, corresponde à estratégia B da Figura 3.4), a questão é verificar se o percentual de CPU solicitado está disponível na porção total da categoria, tal que a soma das porções alocadas a cada thread não exceda esse total.

3.3.3 Elementos do Framework para Sintonização de QoS

O framework para sintonização de QoS especializado para a arquitetura QoSOS é ilustrado pela Figura 3.11. O mecanismo de monitoração é modelado pela classe `QoSOSMonitor`, que define o método `getStatistics()`. Esse método é responsável por obter as medições que descrevem os parâmetros reais de QoS correntemente fornecidos aos usuários. Uma instância da classe `MonitoringStrategy` é a responsável pela realização desses cálculos. Com o uso do pattern Strategy, a arquitetura permite que várias estratégias sejam adotadas para a verificação do desempenho dos fluxos, de acordo com os parâmetros de caracterização que os regem. Para que seja possível a monitoração individual dos fluxos, uma instância de `QoSOSMonitor` deve estar associada a cada fluxo, por isso existe o relacionamento de agregação entre `QoSOSTuner` (o agente de sintonização) e `QoSOSMonitor`.

A sintonização de QoS em sistemas operacionais é centralizada, uma vez que uma instanciação da classe `QoSOSTuner` tem a capacidade de identificar todos os recursos que estão envolvidos em uma provisão de QoS corrente. Caso os valores medidos para um dado fluxo indiquem que seu contrato de serviço está sendo violado por qualquer das partes, o monitor gera um alerta para o agente de sintonização, utilizando o método `alert()`. Já que o agente de sintonização do sistema operacional é responsável pela gerência dos recursos, o seu método

`tune()` pode ser acionado em resposta ao alerta. O agente se vale das estratégias de sintonização (classe `TunningStrategy`) para definir, através do método `calcResponsibilities()` quais recursos estão envolvidos na provisão de QoS ao fluxo identificado, de forma análoga à divisão de responsabilidades presente na negociação de QoS.

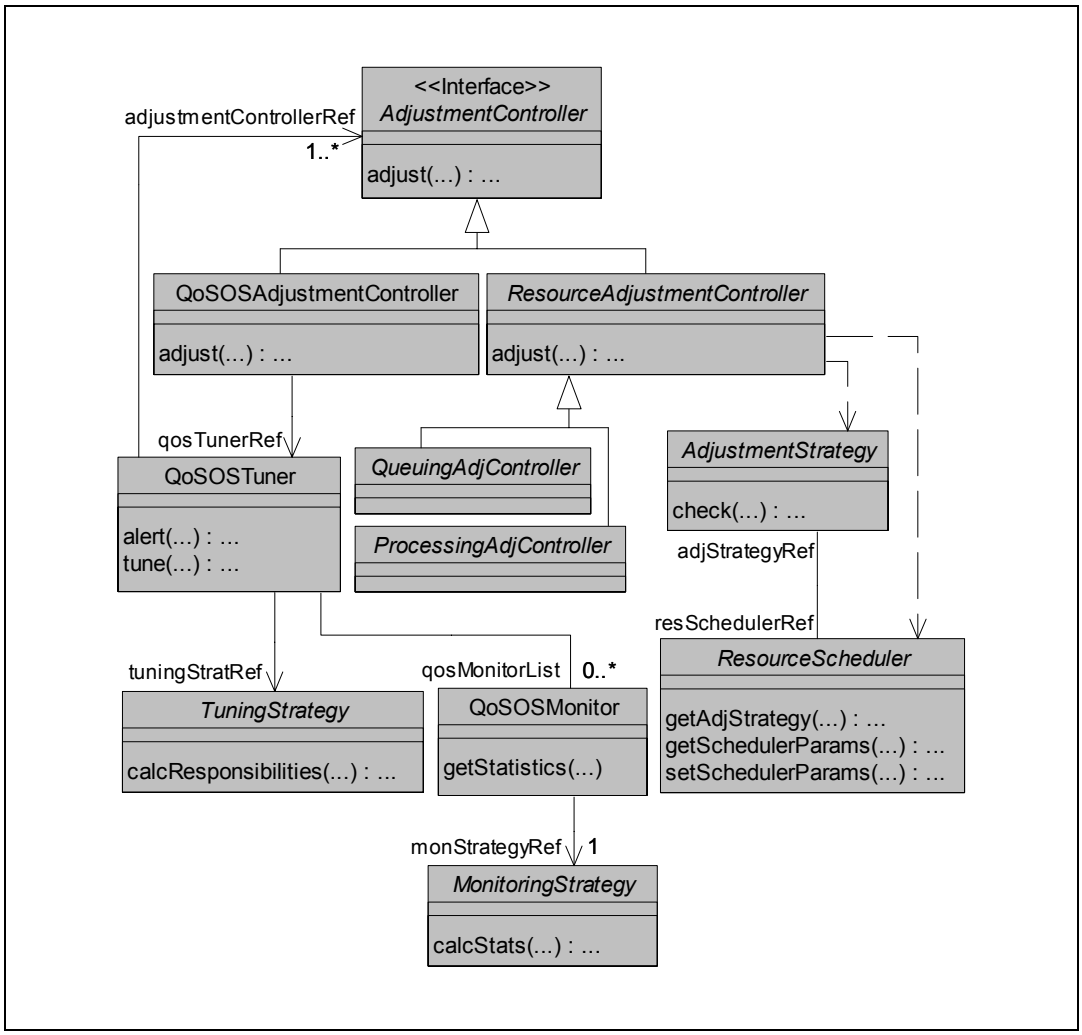

Figura 3.11 - Framework para Sintonização de QoS

O controlador de ajuste do sistema operacional (classe `QoSOSAdjustmentController`) deve ter o método `adjust()` invocado por sintonizadores de níveis de abstração superiores, que lá identificaram violações no contrato do serviço. Dessa forma, uma requisição de ajuste é repassada pela instanciação de `QoSOSAdjustmentController` ao sintonizador QoSOS (classe `QoSOSTuner`).

Os controladores de ajuste referentes a cada recurso real envolvido na sintonização (classe `ResourceAdjustmentController`) são acionados, por meio do método `adjust()`, para efetuarem as modificações nos escalonadores com o objetivo de atender o fluxo corretamente. As instâncias da classe `ResourceAdjustmentController` fazem uso de estratégias para verificar, de forma efetiva, a viabilidade de aplicação de novos parâmetros no escalonamento do recurso virtual já criado. Com esse intuito, as especializações da classe `AdjustmentStrategy` definem o método `check()`. O pattern Strategy foi novamente utilizado para que várias estratégias possam ser implementadas, conforme os parâmetros de caracterização das categorias de serviço disponibilizadas. Caso as estratégias utilizadas confirmem a viabilidade de ajuste, os próprios controladores de admissão solicitam as modificações aos escalonadores de recursos correspondentes (classe `ResourceScheduler`), utilizando o método `setSchedParams()`.

3.3.4 Exemplo de Aplicação do Framework para Sintonização de QoS

O exemplo ilustrado pela Figura 3.12 instancia um mecanismo para pequenos ajustes nos escalonadores de CPU, dado um monitoramento gerado externamente ao sistema operacional. Ao ser detectada uma degradação da QoS em um fluxo de instruções (e.g. um servidor de vídeo não consegue gerar os quadros na taxa média requerida), o mecanismo de sintonização do sistema operacional pode ser acionado para que os ajustes necessários no escalonamento da thread possam ser feitos.

A classe `QoSOSAdjustmentController` é o controlador de ajuste do sistema operacional, responsável por receber do sintonizador de um nível de abstração superior a requisição de ajuste nos recursos gerenciados pelo sistema operacional (método `adjust()`). A classe `QoSOSTuner` representa o agente de sintonização, invocado pelo controlador de ajuste do sistema pelo método `tune()`. A implementação desse método deve requisitar a uma instância da classe `CPUAdjustmentController` a execução de ajustes, através do método `adjust()`, necessários para que a thread possa operar normalmente. As

estratégias implementadas por `SoftRTThreadAdjStrategy` e `HardRTThreadAdjStrategy` verificam a viabilidade de o referido escalonador ter seus parâmetros ajustados sem violar outras reservas já garantidas. Se a verificação obteve sucesso, o controlador de ajuste, ainda no contexto do método `adjust()`, solicita a uma instância da classe `CPUScheduler` (correspondente à categoria de serviço da thread) a modificação dos parâmetros por meio do método `setSchedParams()`.

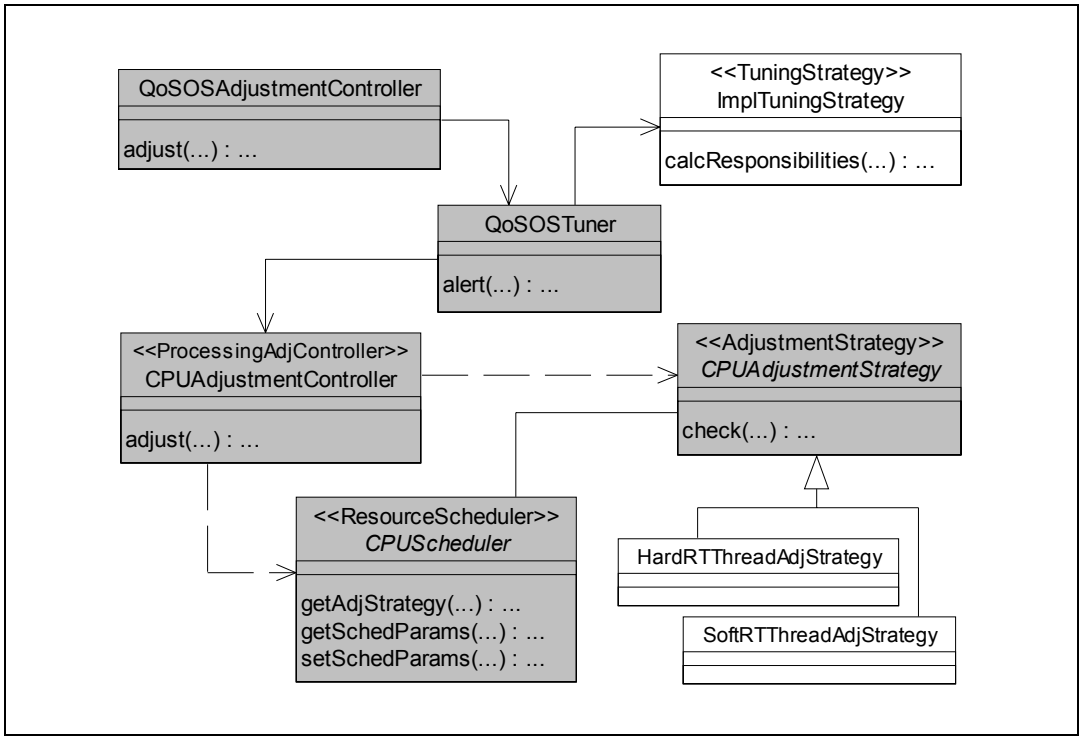

Figura 3.12 - Exemplo de aplicação do Framework para Sintonização de QoS

3.4 Adaptação de Serviços

Embora os frameworks genéricos para provisão de QoS ofereçam a projetistas de sistemas o conceito de *hot-spots* específicos de serviço, permitindo a modelagem de sistemas adaptáveis à introdução de novos serviços, tais *hot-spots* apresentam relações indiretas de dependência entre si, que dificultam a manutenção da consistência do sistema face a adaptações. Nesse contexto, a implementação de "meta-mecanismos" que automatizem a adaptação do sistema a novos serviços ou a novas políticas de provisão de QoS, e que observem questões como manutenção de consistência e restrições de reconfiguração relacionadas a

segurança, é altamente desejável. O framework para adaptação de serviços foi elaborado neste trabalho para preencher parte dessa lacuna deixada pelos frameworks genéricos para provisão de QoS, em uma abordagem específica para sistemas operacionais. Todas as formas de adaptação abordadas no Capítulo 2 foram consideradas na construção do framework, no intuito de torná-lo o mais genérico possível.

As ações de adaptação requeridas pelos administradores do sistema, ou por um meta-mecanismo externo (por exemplo, no caso de redes programáveis, um protocolo de sinalização aberto (Lazar, 1997), ou outro mecanismo de gerência), devem ser controladas por um gerente de adaptação, responsável por receber as requisições, fazer verificações sobre a possibilidade de aceitação, inserir ou substituir o componente alvo e, finalmente, atualizar as referências nos mecanismos a ele relacionados. Para a criação de um serviço inteiramente novo, todos os componentes que implementam as políticas de provisão de QoS devem ser fornecidos ao gerente, juntamente com a localização da nova categoria na hierarquia de categorias de serviço.

Parte dos testes a que se refere o parágrafo anterior compreende a verificação de segurança da inserção do componente, que é delegada pelo gerente de adaptação a um agente específico. De um modo geral, o agente de verificação de segurança deve analisar cada novo componente levando em conta os seguintes aspectos básicos:

- *Confiabilidade*. O fornecedor do componente deve ser confiável.
- *Restrição de contexto*. As ações descritas pelo componente devem estar restritas ao contexto no qual o componente será aplicado.
- *Isolamento*. As ações descritas pelo componente, se estiverem logicamente erradas, não podem prejudicar a provisão de serviços de outras categorias ou a operação de outros subsistemas.

Além disso, pode ser que já exista um componente instalado no sistema com a mesma funcionalidade do componente submetido ou que as estruturas que o utilizariam não mais estão presentes, anulando sua utilidade. Essas verificações

ficam a cargo do agente de sondagem e devem ser requisitadas pelo gerente de adaptação antes da instalação do componente.

Se as verificações foram bem sucedidas, o gerente de adaptação submete a implementação do componente à *porta de adaptação* a ela correspondente. Portas de adaptação são as estruturas existentes no sistema operacional responsáveis por disponibilizar a implementação do componente aos mecanismos que a utilizarão. Exemplos de portas de adaptação são as tabelas LDS e o subsistema de módulos de *kernel* do Linux. Por fim, o gerente atualiza as estruturas que devem fazer referência ao novo componente, como um escalonador faz a um componente de criação ou a uma estratégia de escalonamento, entre outros.

Um último detalhe importante a ser observado está na remoção de componentes do sistema. Além da verificação de segurança, que confirma se o solicitante está autorizado para a ação, um outro teste, chamado de verificação de consistência, deve ser executado. O teste de consistência da remoção tem a responsabilidade de verificar se um componente, ao ser removido do sistema, não acarretará no mau funcionamento ou total parada de outros componentes. Uma estrutura de dependências deve, então, ser mantida no sistema para que essa checagem possa ser feita.

Nota-se que a funcionalidade dos mecanismos de adaptação pode ser aplicada não somente à infra-estrutura de provisão de qualidade de serviço, como também para outras partes do sistema operacional, como o subsistema de rede (pilha de protocolos), o gerenciamento de drivers, o sistema de arquivos, entre muitos outros. Obviamente, essa capacidade deve ser provida pelo *kernel*, atribuindo a esses subsistemas o suporte a portas de adaptação.

3.4.1 Elementos do Framework para Adaptação de Serviços

A Figura 3.13 ilustra o framework para adaptação de serviços. A classe `AdaptationManager` representa o gerente de adaptação, disponibilizando uma interface única de solicitação de adaptações, por meio dos métodos `createService()` para a criação de todo um novo serviço, e

`setComponent()` para a substituição de um único componente. O método `createService()` recebe como parâmetros a categoria de serviço logo acima na hierarquia (se houver) e a lista de componentes que implementam todas as políticas de provisão de QoS. O método `setComponent()` tem como parâmetros a categoria de serviço referente ao componente e o próprio componente.

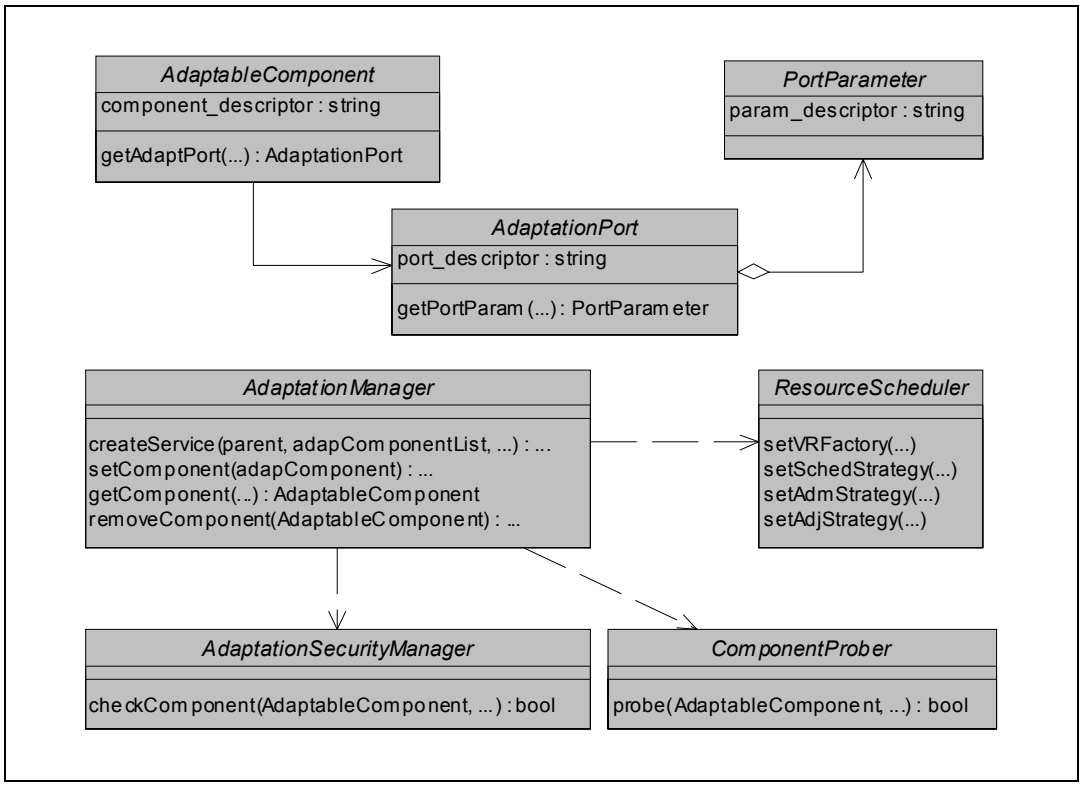

Figura 3.13 - Framework para Adaptação de Serviços

A classe `AdaptationSecurityManager` representa o agente de verificação de segurança, cuja responsabilidade está implementada através do método `checkComponent()`, a ser utilizado pelo gerente de adaptação instanciado. O agente de sondagem é modelado por uma instância da classe `ComponentProber` que implementa o método `probe()` para a verificação da necessidade de instalação do componente. A interação do gerente de adaptação com os mecanismos que utilizarão os componentes está exemplificada pelo relacionamento de dependência entre `AdaptationManager` e os escalonadores de recursos (classe `ResourceScheduler`)[16]. As referências dos escalonadores a

[16] Obviamente, outros mecanismos devem ser alvo de adaptações, e portanto estarão sujeitos à atuação do gerente de adaptação, como os mapeadores, negociadores, agentes de sintonização e monitores. A implementação desses relacionamentos é delegada a trabalhos futuros.

componentes como estratégias de escalonamento, estratégias de admissão e criadores de recursos, devem ser atualizadas conforme o caso de substituição.

A classe `AdaptableComponent` representa um componente de forma abstrata, indicando a função que deve desempenhar uma vez instalado. O método `getAdaptPort()` retorna a porta de adaptação do sistema, modelada pela classe `AdaptationPort`. A classe `PortParameter`, por fim, representa a forma de implementação do componente, ou seja, qual é o meio de representação de sua funcionalidade (e.g. arquivos de código fonte ou objeto, valores de parâmetros, ponteiros para rotinas existentes).

3.4.2 Exemplo de Aplicação do Framework para Adaptação de Serviços

A Figura 3.14 ilustra um exemplo de aplicação do framework para adaptação de serviços, utilizando-se da flexibilidade oferecida pela estrutura LDS hierárquica que já vinha sendo abordada nos outros exemplos. Nesse caso de uso, apenas a substituição da estratégia de escalonamento utilizada por um dos escalonadores de recursos é considerada.

As classes `AdaptableComponent`, `AdaptationPort` e `PortParameter` foram especializadas para as classes `SchedulingStrategyComponent`, `LDSTablePort` e `LDSTableValues`, respectivamente, de forma a abstrair, sob a forma de componente, uma estratégia de escalonamento implementada por valores de uma tabela do algoritmo LDS.

A classe `ImplAdaptationManager` implementa o gerente de adaptação responsável por receber solicitações de adaptação através do método `setComponent()`. De posse da categoria de serviço à qual a estratégia de escalonamento está relacionada, o gerente determina qual o escalonador de recursos (especialização da classe `ResourceScheduler`) deve ter sua tabela LDS modificada. Percebe-se que não houve a necessidade de instanciação de agentes de verificação de segurança nem de sondagem devido às características restritas de utilização dos valores de uma tabela LDS. No próximo Capítulo, será

discutido um caso de adaptação de estratégias de admissão que demanda um cuidado bem maior.

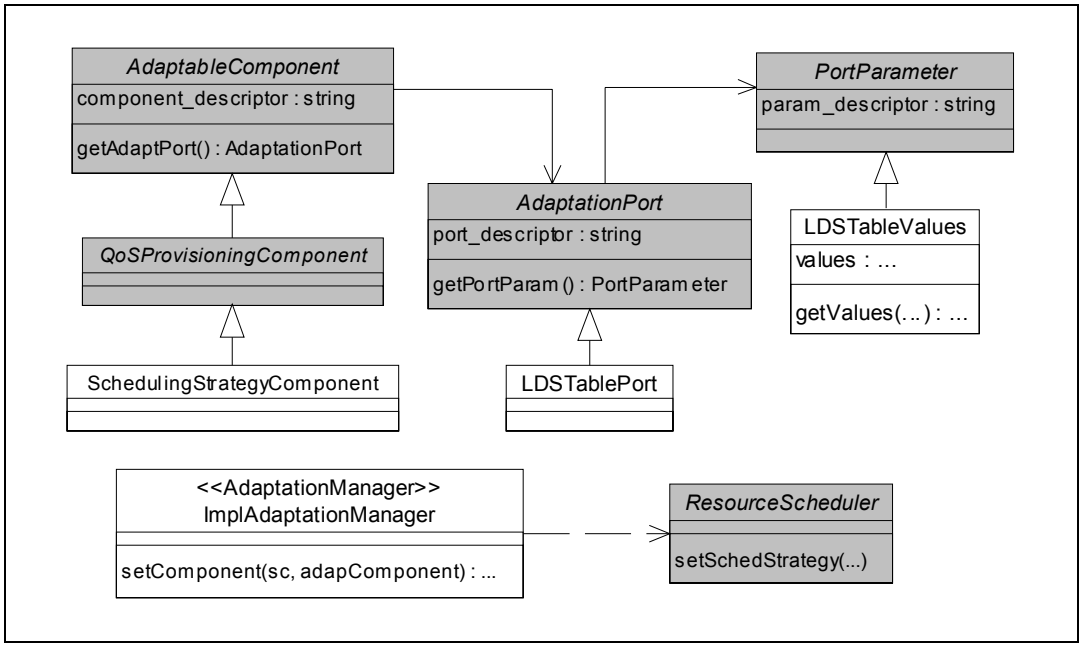

Figura 3.14 - Exemplo de aplicação do Framework para Adaptação de Serviços

3.5 Resumo do Capítulo

No presente Capítulo, a arquitetura adaptável QoSOS foi descrita a partir de um conjunto de frameworks que representam os mecanismos e estruturas para a provisão de QoS em sistemas operacionais. Por meio de exemplos, foi apresentado como esses frameworks podem ser utilizados na modelagem de cenários reais do subsistema de escalonamento de processos de um sistema operacional. A arquitetura descreve não somente a forma como o sistema deve ser configurado para a provisão de serviços com QoS, mas também como ele pode ser alterado em tempo de operação, para que novos serviços possam ser oferecidos conforme a demanda. O framework para Adaptação de Serviços abrange a utilização de, pelo menos, duas formas de adaptabilidade em sistemas operacionais (parâmetros de algoritmos e módulos programáveis do *kernel*), livres de atualizações em hardware ou firmware e de paradas no fornecimento de outros serviços, descritas nas Seções 2.4.3 e 2.6. Esses são requisitos importantes para a definição de sistemas realmente adaptáveis.

4
Exemplo de Aplicação da Arquitetura

Com o objetivo de demonstrar como a arquitetura QoSOS pode ser aplicada na modelagem de um cenário real de provisão de QoS, este Capítulo descreve a implementação do suporte aos serviços integrados (Braden, 1994) sobre o subsistema de enfileiramento de pacotes de rede de um GPOS específico. Além disso, é mostrado como esse mesmo sistema deve ser modificado para aceitar a instanciação do framework de adaptação de serviços e, assim, possibilitar que as estratégias de admissão das categorias de serviço Intserv sejam substituídas em tempo de operação.

O modelo de serviços integrados foi escolhido por já ter sido alvo de outro trabalho produzido no Laboratório TeleMídia da PUC-Rio, que promoveu a definição de uma arquitetura para a provisão de QoS na Internet (Mota, 2001), baseada nos mesmos frameworks genéricos aqui utilizados. Além da própria definição da arquitetura, uma grande contribuição desse trabalho consistiu na construção de uma API de solicitação de serviços independente do modelo ou tecnologia empregada internamente no provedor de serviços. Naquela ocasião, foi proposto um cenário de uso da arquitetura constituído por sub-redes ATM e Ethernet, onde foram modelados os agentes de negociação de QoS do nível de rede associados aos roteadores e estações finais. A reserva de recursos, contudo, era localizada apenas na sub-rede ATM, pois as estações finais e roteadores dependiam de uma implementação de QoS para seus sistemas operacionais. Por isso, o cenário aqui proposto pode ser considerado um complemento daquele, sendo prudente o uso do mesmo modelo de QoS para a solicitação dos serviços a fim de integrar os dois projetos.

Nota-se que a aplicação do framework apenas sobre o enfileiramento de pacotes de rede não implica em uma implementação eficiente de provisão de QoS no domínio de sistemas operacionais, pois, como já discutido, a orquestração deve abranger outros recursos importantes, como a CPU. No entanto, considerar apenas

um dos subsistemas relevantes é suficiente para o propósito de demonstração de como os mecanismos descritos pela arquitetura podem ser aplicados.

Já o desenvolvimento do suporte à adaptação em um GPOS fica limitado, porque não é possível, por exemplo, criar um novo serviço submetendo todas as políticas de QoS sob a forma de componentes adaptáveis. Para permitir essa funcionalidade, seria necessário tornar flexível toda a estrutura de escalonamento dos recursos em questão definida no *kernel* desse GPOS. Assim, optou-se por adicionar ao *kernel* uma função antes ausente e demonstrar que ela pode ser substituída durante a operação do serviço. A extensão de um GPOS para adotar a arquitetura de adaptação de serviços é um trabalho ainda pouco explorado, mas muito interessante e deve incluir a reprogramação de grande parte dos subsistemas mais importantes.

O restante deste Capítulo encontra-se estruturado da seguinte forma. Primeiramente, o cenário implementado é apresentado de modo a estabelecer a infra-estrutura de rede, os equipamentos e as funcionalidades esperadas do sistema operacional utilizado. Em seguida, são relatados os procedimentos que foram necessários para a preparação da infra-estrutura do cenário, como a implementação de interfaces de baixo nível, modificações no *kernel* e configuração do sistema. Por fim, a descrição da instanciação dos frameworks é o assunto abordado, abrangendo detalhes sobre várias fases da provisão de QoS.

4.1 Descrição do cenário

A Figura 4.1 ilustra o cenário de uso, que é composto por várias sub-redes Ethernet interligadas por roteadores, formando um provedor Intserv. Pressupõe-se, ainda, que agentes de negociação RSVP (Braden, 1997) estão implementados em cada estação das sub-redes e nos roteadores.

Tanto as estações finais quanto os roteadores são gerenciados pelo sistema operacional Linux, com a versão 2.2.18 do *kernel*. Essa versão, além de ser muito estável, apresenta mecanismos para a manipulação do enfileiramento de pacotes junto às interfaces de saída para a rede. Denominado como controle de tráfego do

Linux, esse conjunto de mecanismos foi apresentado na Seção 2.5.6 e será utilizado para a reserva de recursos nos buffers de comunicação por ele controlados. Apesar de não ter importância para a descrição da instanciação, cada *kernel* está estendido pela arquitetura LRP descrita na Seção 2.5.5, de forma a tornar o recebimento de pacotes de rede mais eficiente.

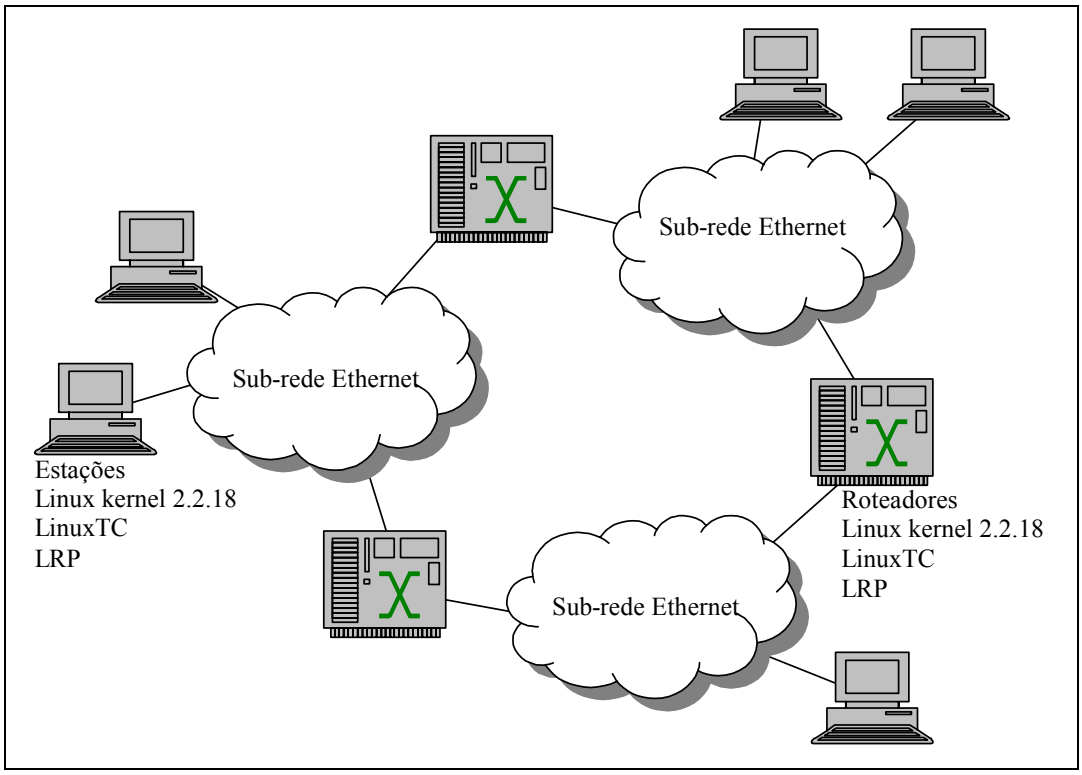

Figura 4.1 - Cenário de provisão de serviços Intserv

4.2 Infra-estrutura desenvolvida

Por intermédio do programa *tc*, o controle de tráfego do Linux (LinuxTC)[17] oferece métodos muito ricos, mas pouco dinâmicos, para a configuração dos seus componentes. Isso porque os desenvolvedores desse subsistema não projetaram uma interface de programação para que aplicações e protocolos de negociação pudessem configurar as características de desempenho da comunicação pela rede.

[17] É necessário configurar e recompilar o kernel para habilitar o LinuxTC, pois a configuração padrão não o inclui. Para isso, podem ser utilizadas as interfaces de configuração *menuconfig* (modo texto) ou *xconfig* (modo gráfico) localizadas junto ao código do kernel, por meio dos comandos *make menuconfig* ou *make xconfig*.

Recentemente, um projeto internacional de código aberto chamado *TCAPI* (Olshefski, 2001) foi criado pela IBM com o objetivo de preencher essa lacuna. Escrita em linguagem C, a versão 1.2 da interface *TCAPI* foi utilizada para a configuração do LinuxTC no cenário. Contudo, essa ferramenta mostrou-se deficiente por não disponibilizar muitas operações importantes oferecidas pelo LinuxTC (criação de filtros RSVP, remoção de qualquer disciplina, "classe" [18] ou filtro) e, ainda, por conter algumas falhas de programação. Devido à facilidade de acesso ao código da interface *TCAPI*, várias modificações puderam ser feitas até que o funcionamento ideal com as funções necessárias fosse atingido. Tais alterações foram submetidas à apreciação do administrador do projeto e todas foram aprovadas e integradas ao software original.

Como discutido na Seção anterior, é inviável a reprogramação de todo o controle de tráfego do Linux para que ele se torne integralmente adaptável e possa, assim, ser utilizado para descrever um exemplo de instanciação do framework de adaptação para a criação de novos serviços. Por isso, adotou-se a tática de se introduzir políticas anteriormente não suportadas pelo LinuxTC no *kernel* do sistema, que fossem adaptáveis desde sua concepção. Uma dessas políticas é o conjunto de estratégias de admissão adaptáveis (framework de negociação de QoS), que, se implementada no *kernel*, pode ser utilizada por controladores de admissão definidos no espaço do usuário (como é o caso) ou no próprio *kernel*. Dessa forma, optou-se por transferir as estratégias de admissão para o espaço do *kernel* através da utilização dos módulos do sistema descritos na Seção 2.6. Essas estratégias devem ser codificadas em um formato bem definido, como se segue (Figura 4.2):

- Função `strat_check()`: é o método acessível pelo controlador de admissão no espaço do usuário para a verificação da viabilidade da admissão em si. O parâmetro é o endereço de memória dos argumentos fornecidos através da interface `ioctl`, cujo conteúdo deve ser copiado para o espaço do *kernel* (função do *kernel* `copy_from_user()`).

[18] O elemento "classe" do LinuxTC aparecerá sempre entre aspas ao longo do texto, para que não seja confundido com as classes do modelo orientado a objetos em que se baseia a modelagem da arquitetura.

```
#include <linux/module.h> /* criacao de modulo */
#include <linux/kernel.h> /* programacao no espaco do kernel */
#include <linux/fs.h>     /* operacoes com arquivo (ioctl) */
#include <asm/uaccess.h>  /* copy_to/from_user */

#include <net/adm_strategy.h> /* administracao das estrategias */

#define STRAT_NAME "Exemplo de estratégia 1.0"

static int example_check(unsigned long);

struct adm_strategy example_strategy = { STRAT_NAME, example_check };

/* Funcoes visiveis pelo kernel, quando registradas via struct */
/* apenas *_check deve estar aqui */
static int example_check(unsigned long arg) {
    int feasible;
    ...
    return(feasible);
}

#ifdef MODULE
/* Funcoes executadas no carregamento e remocao do modulo */
int init_module(void) {
    if (register_strategy(&example_strategy)) {
        /* Erro ao inserir o modulo */
        ...
        return -EIO;
    }
    ...
    return 0;
}

void cleanup_module(void) {
    ...
    if (unregister_strategy(&maxflows_strategy)) {
        /* Erro ao remover o modulo, não há como retornar o erro*/
        ...
    }
}
#endif
```

Figura 4.2 - Modelo de implementação de estratégias de admissão através de módulos em linguagem C

- Função `init_module()`: responsável por realizar quaisquer operações necessárias no momento do carregamento do módulo, como solicitar que a estratégia seja registrada pelo *kernel* (nova função do *kernel* `register_strategy()`, cujo parâmetro é uma estrutura com o nome da estratégia e um ponteiro para a função `check()`). Com o registro da estratégia, o *kernel* se torna capaz de identificar uma chamada `ioctl()` a ela direcionada. Outros exemplos de operações que poderiam ser necessárias durante a inserção de um módulo são a inicialização de variáveis locais e a alocação de memória do espaço do *kernel*.

- Função `cleanup_module(void)`: responsável por executar quaisquer operações necessárias no momento da remoção de um módulo, como

remover o registro da estratégia (nova função do *kernel* `unregister_strategy()`, onde a mesma estrutura acima descrita é fornecida como parâmetro para a identificação da estratégia).

O *kernel* 2.2.18 foi modificado tanto para prover as funções de gerenciamento dos módulos de estratégias de admissão, como para oferecer uma interface de comunicação entre controlador e estratégia por meio de um novo dispositivo[19], batizado como `adm_strat`. O método `check()` das estratégias deve ser invocado pelo controlador por meio de uma chamada `ioctl()` nesse dispositivo. Tal função recebe como parâmetros um código sinalizando o método requerido e a estratégia correspondente, além de um ponteiro para os argumentos do método. Dessa forma, o controlador de admissão terá um trecho de código (em C) semelhante ao da Figura 4.3, ao testar efetivamente a viabilidade da reserva.

```
int feasible = 0;
...
{
      int fd;
      int cmd; /* Código para metodo e estrategia a serem utilizados */
      struct check_parameters params;  /* parametros do metodo check */
      int strat; /* estrategia a utilizar, conforme a
                    categoria de serviço solicitada */
      ...
      cmd = ADM_STRAT_CHECK * ADM_STRAT_FNBASE /* Codificacao de check */
               + strat;         /* Codificacao da estrategia requerida */
      if ((fd = open("/dev/adm_strat", O_RDONLY)) == -1) {
            /* Erro ao acessar o dispositivo */
            ...
      }
      else {
            /* Chamada da funcao check da estratégia no kernel,
               de forma codificada */
            feasible = ioctl(fd, cmd, &params);
            ...
      }
      ...
}
...
if (feasible) {
      /* Pode ser gerada uma pre-reseva */
}
...
```

Figura 4.3 - Trecho de código de um controlador de admissão para o acesso aos módulos de estratégias de admissão

As novas funcionalidades foram incluídas no utilitário de configuração do *kernel*, na porção referente ao LinuxTC:

[19] No Linux, dispositivos são tratados como arquivos, podendo ser acessados por chamadas como `open()`, `read()` e `ioctl()`.

```
[*] Flow Admission Control Strategies (NEW)
      (6) Maximum number of concurrent strategies (NEW)
      < > SimpleSum Admission Strategy (NEW)
      < > MaxUsers AdmissionStrategy (NEW)
```

4.3 Instanciação da arquitetura

A instanciação da arquitetura QoSOS sobre o cenário de uso proposto levou ao desenvolvimento de duas interfaces de programação para aplicações (API) em Java, através das quais foram modelados muitos dos mecanismos descritos. A primeira API, denominada *QueuingQoS*, oferece a solicitação de serviços com QoS por parte dos agentes de negociação de rede, promovendo controle de admissão e criação de recursos virtuais sobre o subsistema de enfileiramento de pacotes de rede. A segunda API, denominada *AdaptQoS*, permite que ações de adaptação sejam requisitadas pelos administradores de sistemas operacionais especificamente sobre as categorias de serviço acessíveis pela interface *QueuingQoS*.

A Figura 4.4 permite uma visão geral dessa instanciação, ilustrando como os componentes da interface *QueuingQoS* interagem e modificam a estrutura do controle de tráfego, ações essas demarcadas por linhas tracejadas. A interface *AdaptQoS*, por sua vez, tem seu fluxo de controle caracterizado pelas linhas pontilhadas, agindo apenas sobre a estrutura adaptável de estratégias de admissão. O fluxo dos pacotes de rede entre as filas do controle de tráfego está representado pelas linhas contínuas.

4.3.1 Iniciação do sistema

As tarefas pertinentes à fase de iniciação do sistema estão implementadas por um programa em separado, acionado no momento da inicialização do sistema (boot). Ele é capaz de montar a estrutura inicial da árvore de recursos virtuais sobre as filas do nível de enlace (denominação dada às filas controladas pelo LinuxTC) que fazem parte do subsistema de rede das estações. As parcelas de alocação da largura de banda dessas filas associadas às categorias de serviço

(inclusive melhor esforço, que não tem qualquer garantia negociada) são passadas pela linha de comando do programa.

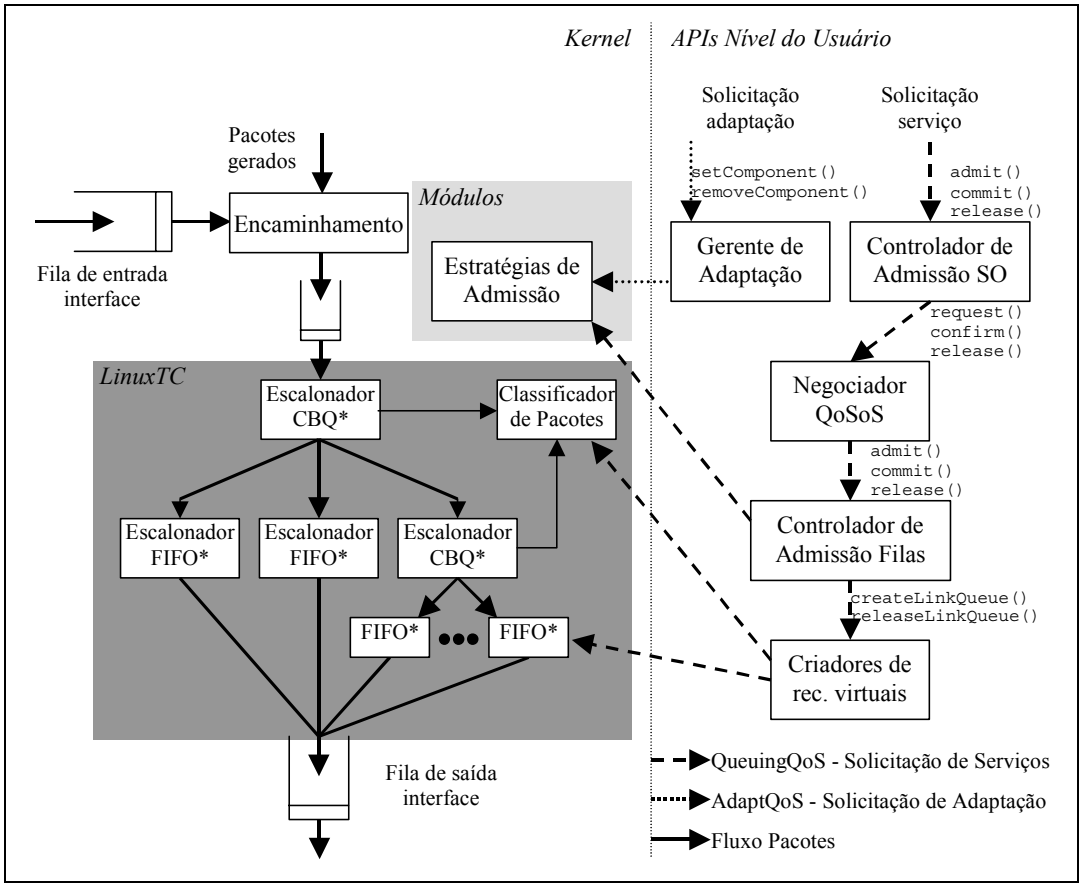

Figura 4.4 - Visão geral da implementação

A Figura 4.5 ilustra um exemplo de configuração inicial da árvore de recursos virtuais, onde o escalonador raiz utiliza a estratégia de escalonamento CBQ (Floyd, 1995), capaz de atribuir parcelas de largura de banda aos seus escalonadores filhos. A categoria de serviço garantido, detentora de 30% da capacidade da fila de enlace, também foi associada à estratégia CBQ, para redistribuir sua largura de banda entre seus fluxos. A categoria de carga controlada tem disponível uma parcela de 50% do enlace e utiliza a estratégia FIFO para o escalonamento dos pacotes, já que não fornece garantias severas de alocação de recursos. Por último, a categoria de melhor esforço será utilizada pelos fluxos que não solicitaram a provisão de serviços com QoS, que podem juntos utilizar apenas 20% do recurso. As estratégias de admissão relacionadas a cada categoria de serviço serão descritas posteriormente, na Seção 4.3.5.

Nota-se que o programa de iniciação do sistema poderia estar utilizando um gerenciador de adaptabilidade para a criação dos serviços, mas não foi assim feito devido à necessidade de suporte à adaptação por parte de todas as políticas de QoS distribuídas entre o LinuxTC e a interface do usuário.

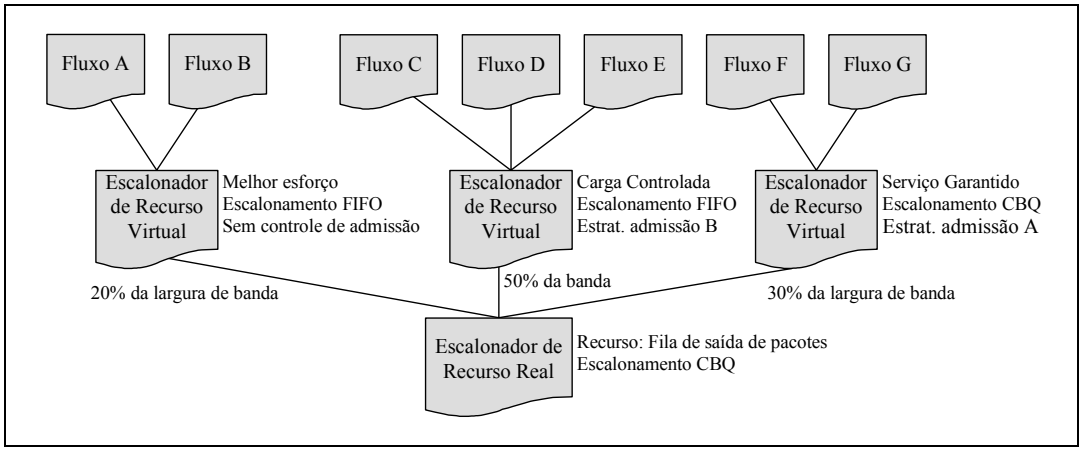

Figura 4.5 - Árvore de recursos virtuais da fila de pacotes

4.3.2 Parametrização de serviços

O framework de parametrização de serviços foi instanciado seguindo a proposta de (Mota, 2001), com o objetivo de manter a coerência necessária entre as duas implementações. A Figura 4.6 ilustra a hierarquia de serviços e de parâmetros originada pela especialização do referido framework.

As categorias de serviço garantido e de carga controlada são representadas pelas classes `GuarServiceCategory` e `CLServiceCategory`, respectivamente. O parâmetro RSpec (reservation specification) deve ser adicionado apenas a objetos da classe `GuarServiceCategory` por denotar os requisitos de qualidade a serem reservados pelo sistema, compreendendo a taxa de dados (R, em bytes/s) e o termo de folga (s, referente à diferença entre o retardo obtido a partir da reserva de R e o retardo máximo permitido, em milissegundos). Já o parâmetro TSpec (traffic specification) é de comum utilização por ambas categorias e descreve a caracterização do tráfego gerado pelo o usuário. O valor de r corresponde à taxa de dados em bytes/s, b é o tamanho do bucket em bytes, p é a taxa de pico em bytes/s, m denota o tamanho mínimo da unidade policiada em bytes e M é o tamanho máximo do pacote em bytes.

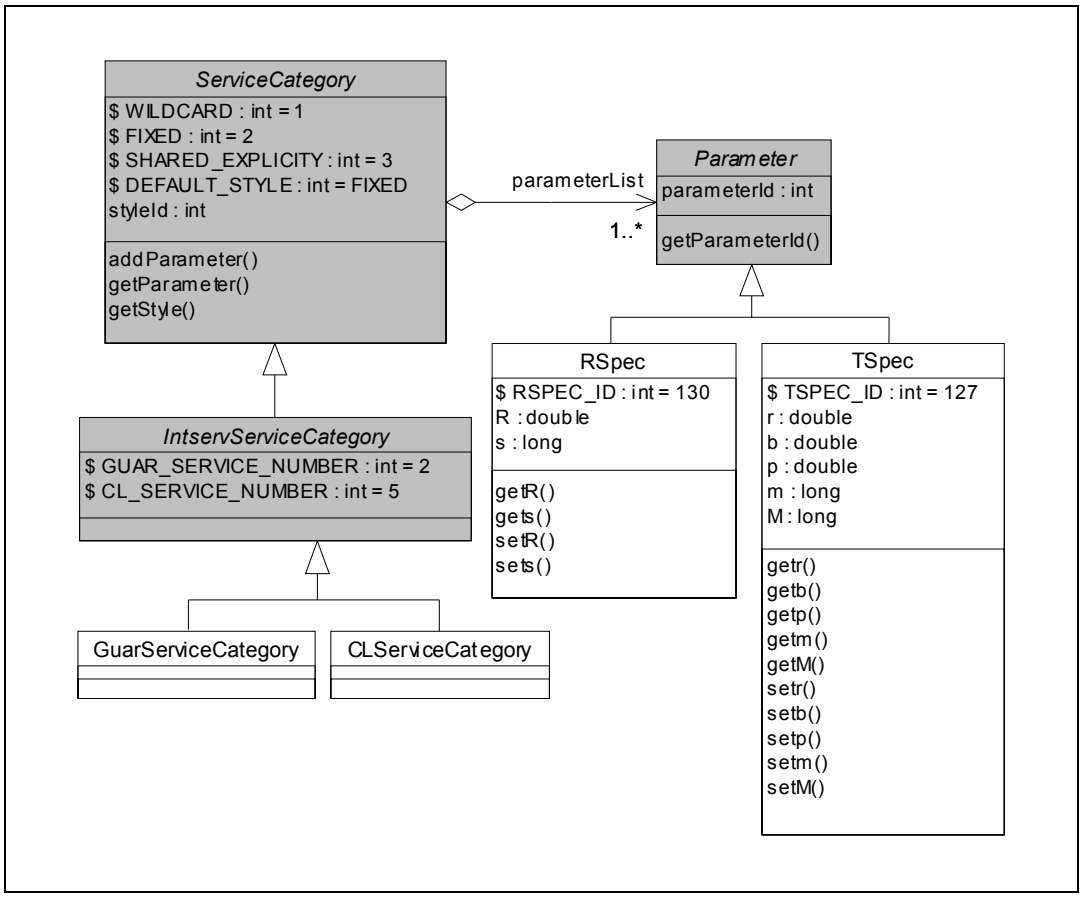

Figura 4.6 - Aplicação do framework para Parametrização de Serviços

Não há a necessidade de definição de outros parâmetros de maior relacionamento com o recurso a ser alocado, já que o algoritmo CBQ e os policiadores do LinuxTC são configurados por valores de mesmo significado que aqueles definidos pelas estruturas RSpec e TSpec.

4.3.3 Compartilhamento de recursos

Os mecanismos do controle de tráfego do Linux puderam ser diretamente modelados pelo framework de escalonamento de recursos. A instanciação da arquitetura ilustrada pela Figura 4.7 reflete, inclusive, os nomes dos métodos encontrados internamente no LinuxTC, na forma como foram definidos em (Almesberger, 1999).

A classe `DeviceQueuingDiscipline` representa a disciplina de enfileiramento raiz associada a uma interface de rede e responsável por receber os

comandos `wakeup()` para iniciar a seqüência de escalonamento de pacotes (função `dequeue()`). Todas as outras disciplinas relacionadas com a mesma interface são modeladas pela classe `InnerQueuingDiscipline` e também utilizam a função `dequeue()` para dar prosseguimento ao escalonamento.

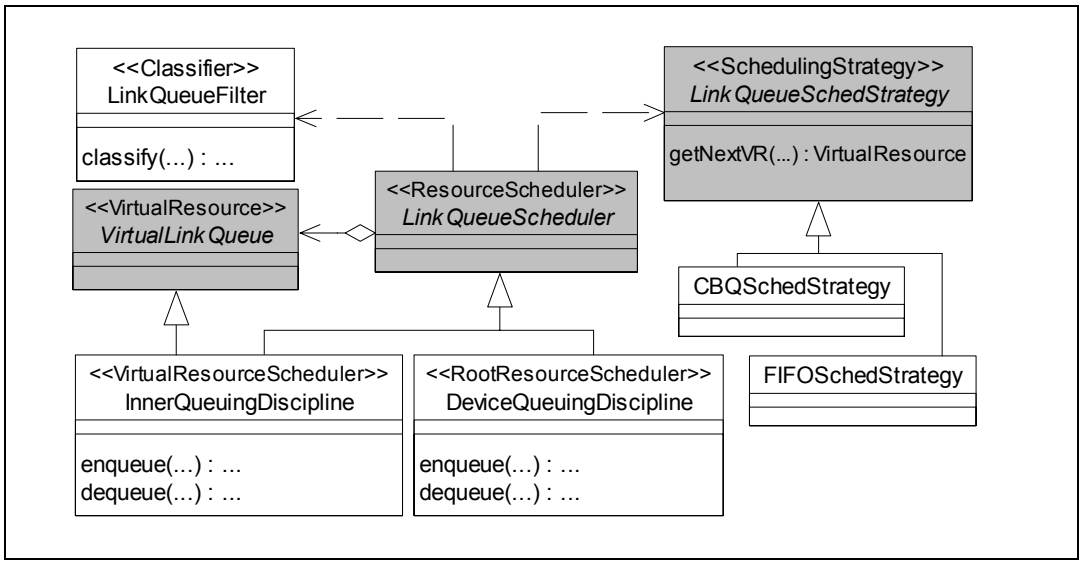

Figura 4.7 - Aplicação do framework de escalonamento de recursos

Quando um pacote é totalmente processado pela pilha de protocolos de rede, é colocado à disposição da disciplina de enfileiramento associada à interface de saída selecionada pelos procedimentos de encaminhamento. A seguir, essa disciplina raiz deve submeter o pacote à análise dos filtros classificadores, modelados por `LinkQueueFilter`, que identificam a "classe" a que pertence o pacote. A disciplina raiz, então, invoca o método `enqueue()` da disciplina folha da "classe" indicada. A seqüência de métodos `enqueue()` continua até que seja encontrada uma disciplina sem "classes", como as disciplinas regidas pela estratégia FIFO.

Nota-se que o elemento "classe" representa apenas uma conexão entre as disciplinas, definindo a parcela de alocação atribuída a cada uma delas. Se uma "classe" não possui uma disciplina folha, a estratégia FIFO é considerada como sua disciplina. As estratégias de escalonamento oferecidas pelo LinuxTC que são utilizadas na provisão de serviços integrados estão representadas na modelagem pelas classes `CBQSchedStrategy` e `FIFOSchedStrategy`.

Já o elemento filter do LinuxTC difere um pouco do mecanismo de classificação modelado pela arquitetura QoSOS, uma vez que ele agrega as funções de classificação e policiamento, como será visto posteriormente. Ao mesmo tempo, cada filtro possui uma única regra para a classificação e um único perfil de tráfego para policiamento.

A seqüência de chamadas `dequeue()` define o escalonamento de cada pacote armazenado em todas as disciplinas. Ao receber a sinalização pelo método `wakeup()`, a disciplina raiz seleciona, de acordo com sua estratégia, aquela "classe" que deve ser atendida e faz a chamada da função `dequeue()` para a disciplina folha correspondente. Quando essa seqüência chega a uma disciplina sem "classes", imediatamente é selecionado o pacote a ser entregue ao driver da interface.

Uma característica importante do LinuxTC é que os pacotes não são copiados de uma disciplina para a outra, como se estivessem sendo transmitidos entre filas. Na realidade, cada disciplina possui uma fila de ponteiros para os *skbufs*[20], os quais permanecem armazenados nos mesmos endereços durante todo o processo de enfileiramento.

Entre os mecanismos modelados pela instanciação do framework de alocação de recursos, ilustrada pela Figura 4.8, somente os componentes de criação de recursos virtuais não estão presentes no subsistema de controle de tráfego do Linux e precisaram ser implementados no espaço do usuário, como parte da API de solicitação de serviços. Foram instanciados dois componentes de criação de modo a corresponderem cada um a uma categoria de serviço específica, já que as necessidades de reserva entre elas são diferentes.

A alocação de recursos para os fluxos que requerem a categoria de serviço garantido é realizada por um objeto da classe `GuaranteedLQFactory`, através do método `createGuarLink()`. Essa operação pode ser dividida em duas etapas: a criação dos filtros de classificação e policiamento e a reserva efetiva da

[20] Skbufs são os buffers de memória para armazenamento dos pacotes de rede no espaço de endereçamento do kernel, muito semelhantes aos mbufs utilizados em outros sistemas operacionais.

largura de banda solicitada. Um único elemento "filtro" do LinuxTC realiza tanto o casamento de sua regra com os dados contidos no cabeçalho, como o policiamento dos pacotes casados seguindo sua regra de perfil de tráfego. Assim, o componente de criação da categoria de serviço garantido solicita a criação de um filtro de classificação e policiamento (através da interface TCAPI, pois o objeto está definido no espaço do usuário) pela função `change()` de um objeto da classe `LinkQueueFilter`. A reserva da largura de banda é feita pela criação de mais uma "classe" junto à disciplina de enfileiramento encarregada de escalonar os pacotes da categoria de serviço garantido. Isso é feito (novamente por intermédio da interface TCAPI) através da função `change()` presente nas especializações da classe `LinkQueueScheduler`.

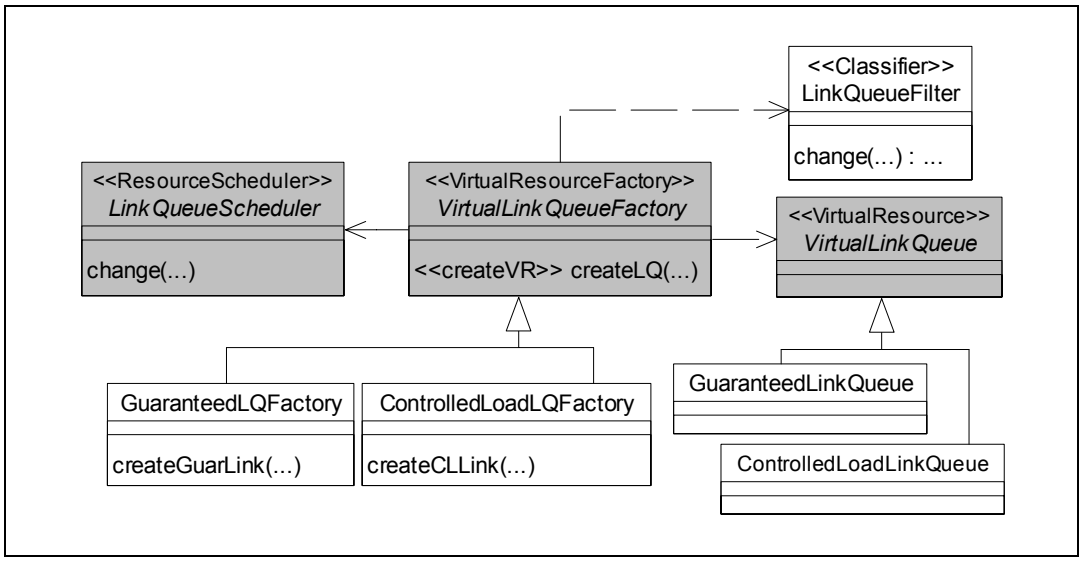

Figura 4.8 - Aplicação do framework para Alocação de Recursos

Com relação ao componente de criação de recursos virtuais associado à categoria de serviços de carga controlada, é suficiente que o filtro de classificação e policiamento seja configurado, pois não existe a reserva efetiva de largura de banda para fluxos dessa categoria.

4.3.4 Solicitação de serviços

O processo de solicitação de serviços ao subsistema de enfileiramento de pacotes do sistema operacional somente acontece depois que os agentes de negociação de rede estabeleceram as parcelas de participação na orquestração de

recursos de cada estação envolvida. Por isso, a API *QueuingQoS* disponibiliza os métodos públicos do controlador de admissão QoSOS para que os agentes de negociação do nível de rede possam solicitar serviços com QoS. O controlador de admissão QoSOS aqui instanciado provê apenas uma interface entre o usuário, o negociador QoSOS e o controlador de admissão das filas de comunicação. Conforme proposto para este cenário de uso, nenhum tipo de orquestração de recursos interna ao sistema operacional é realizada pelo negociador.

As primitivas que compõem a API *QueuingQoS* estão especificadas na Tabela 4.1. As classes `QoSSession` e `FilterSpec` foram definidas por (Mota, 2001) e reaproveitadas, sendo que a primeira representa as informações sobre o destino de um fluxo e a segunda é a identificação da origem de um fluxo.

Solicitação de Serviço `long admit(QoSSession, InetAddress, FilterSpec, ServiceCategory)`	
`QoSSession`	Sessão de QoS
`InetAddress`	Endereço da interface local a ter recursos reservados
`FilterSpec`	Identificação do fluxo especificado
`ServiceCategory`	Categoria de serviço especificada
`Retorno`	Se maior que 0, identificador para a pré-reserva feita pelo controlador de admissão, caso contrário significa inviabilidade ou falha da solicitação
Confirmação de Serviço `boolean commit(long)`	
`Long`	Identificador da pré-reserva a ser confirmada
`Retorno`	Sucesso (true) ou falha (false) da confirmação
Encerramento do Serviço `boolean release(QoSSession, InetAddress, FilterSpec, ServiceCategory)`	
`QoSSession`	Sessão de QoS
`InetAddress`	Endereço da interface local a ter recursos liberados
`FilterSpec`	Identificação do fluxo especificado na solicitação
`ServiceCategory`	Categoria de serviço especificada na solicitação
`Retorno`	Sucesso (true) ou falha (false) da liberação dos recursos

Tabela 4.1 - Primitivas da API de solicitação de serviços

O método `release()` é o único que não foi mencionado na descrição da arquitetura. A partir dele, o usuário solicita a liberação de todos os recursos alocados para a provisão de QoS sobre o fluxo identificado pelos parâmetros `QoSSession` e `FilterSpec`. O retorno informa se a operação foi bem sucedida ou não.

4.3.5
Estabelecimento de contratos de serviço

A Figura 4.9 ilustra a instanciação do framework de negociação de QoS para o cenário proposto. Para atender uma solicitação de serviço feita por meio da primitiva `admit()`, o controlador de admissão do sistema operacional (classe `QoSOSAdmissionController`) encaminha os parâmetros fornecidos ao negociador de QoS (classe `QoSOSNegotiator`). O negociador deve repassar os parâmetros solicitados ao controlador de admissão das filas de enlace (classe `LinkQueuingAdmissionController`, por meio do método `admit()`). Isso pode ser feito diretamente, como dito anteriormente.

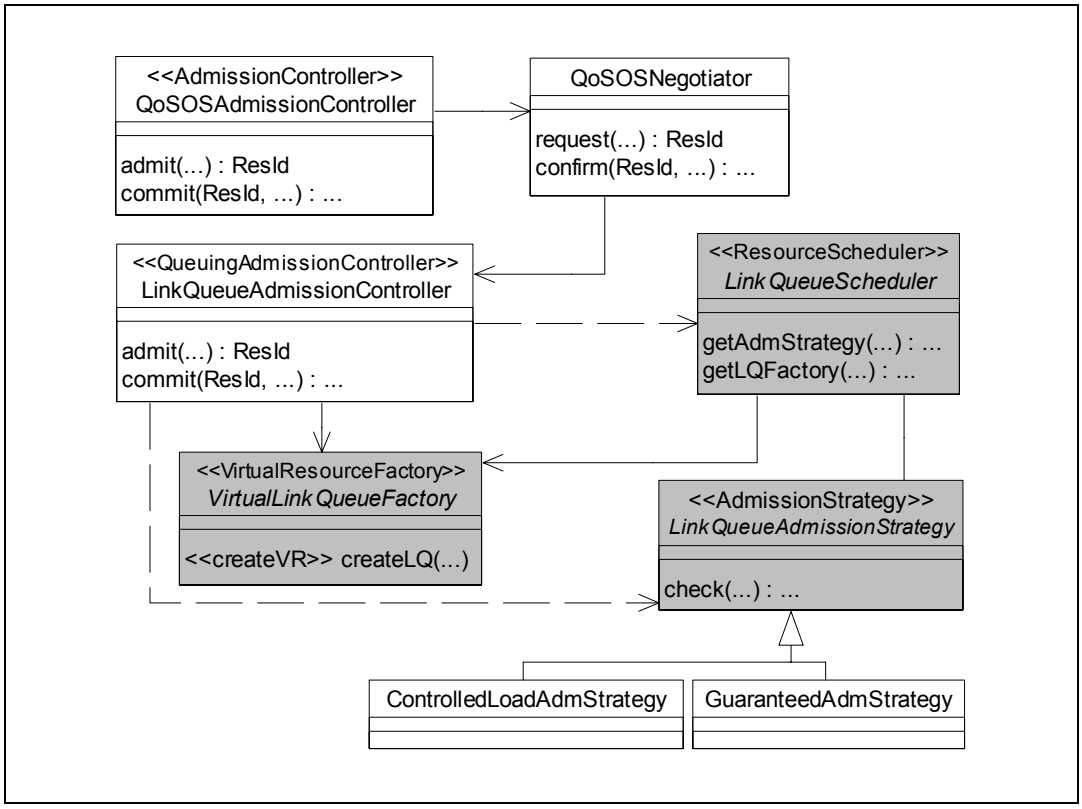

Figura 4.9 - Aplicação do framework para Negociação de QoS

O controlador de admissão das filas verifica a categoria de serviço solicitada e invoca o método `check()` da estratégia de admissão correspondente (classe `LinkQueueAdmissionStrategy`). Esse método analisa os atuais parâmetros de desempenho dos recursos e os compara com a necessidade de QoS solicitada. Se ficar concluído que a solicitação é viável, o controlador de admissão é informado e gera um identificador de pré-reserva, que será utilizado posteriormente na confirmação do serviço. O controlador de admissão das filas retorna o

identificador gerado ao negociador, que repassa ao controlador de admissão QoSOS.

Com o objetivo de confirmar o serviço, o controlador de admissão QoSOS deve invocar o método `confirm()` do negociador, informando o identificador da pré-reserva, que será repassado pelo negociador ao controlador de admissão das filas, utilizando o método `commit()`. Se o identificador ainda for válido, os dados da pré-reserva são recuperados e o componente apropriado de criação de recursos virtuais (uma especialização da classe `LinkQueueFactory`) é acionado, como descrito no framework de alocação de recursos. Essa seqüência de chamadas é ilustrada resumidamente pela Figura 4.4 apresentada anteriormente.

É interessante observar que um trecho do código do controlador de admissão implementado é executado periodicamente para a verificação da validade dos identificadores de reserva. Quando é constatada a expiração de um identificador, todos os dados sobre a reserva são removidos e não mais considerados para a admissão de solicitações futuras.

As estratégias de admissão disponíveis inicialmente correspondem às classes `GuaranteedAdmStrategy` e `ControlledLoadAdmStrategy`. Para que a admissão de um fluxo de serviço garantido seja aprovada, a taxa de dados (R) deve estar disponível no escalonador de recurso virtual. Essa estratégia é conhecida como soma simples e corresponde à estratégia A, especificada na estrutura da árvore de recursos virtuais, apresentada pela Figura 4.5. Já a admissão para carga controlada é regida pela estratégia B da figura, que consiste em testar se a soma dos parâmetros r de todos os fluxos anteriormente admitidos mais o parâmetro r do fluxo que solicitou o serviço não excede a largura de banda alocada à categoria de serviço. Essa estratégia é equivalente à soma simples, mas leva em conta os parâmetros de caracterização do tráfego.

4.3.6 Manutenção de contratos de serviço

Durante a fase de manutenção de serviço, os filtros de policiamento do controle de tráfego do Linux agem de modo a identificar os fluxos submetidos

pelos usuários que excedem a caracterização de tráfego informada no momento da solicitação do serviço. Os filtros são capazes de identificar os pacotes não-conformes e, imediatamente, tomar uma das três decisões, configuradas no momento da criação do filtro: ignorar a notificação e enviar o pacote; promover uma reclassificação do pacote, por exemplo, para uma "classe" de menor prioridade; ou simplesmente descartá-lo.

4.3.7 Adaptação de serviços

A Figura 4.10 ilustra a instanciação do framework de adaptação de serviços sobre a infra-estrutura desenvolvida dentro do *kernel*, descrita anteriormente. A classe `LinuxQoSAdaptationManager` representa o gerente de adaptação implementado no espaço do usuário e sua funcionalidade está restrita à inserção e substituição de módulos que implementam estratégias de admissão. Tais módulos são abstraídos sob a forma de componentes adaptáveis através das classes `AdmissionStrategyComponent`, `KernelModulePort` e `ObjectFile`.

Quando o administrador do sistema ou qualquer outro mecanismo de gerência autorizado solicita a inserção/substituição (`setComponent()`) de um componente através da API *AdaptQoS*[21], o gerente de adaptação instanciado se certifica de que a porta de adaptação é o subsistema de módulos do *kernel*, para o qual será submetida a implementação do componente, por meio da chamada `ins_mod`. Esse subsistema possui uma verificação simples de segurança, modelada pela classe `LinuxAdaptationSecurityManager`, que consiste na autenticação das capacidades atribuídas ao solicitante, seja ele um usuário ou um programa (`função capable()`). Além disso, uma função de sondagem (`modprobe()`) pode ser explorada pela classe `LinuxComponentProber`, mas não foi utilizada na implementação.

A associação entre as categorias de serviço e as estratégias de admissão é dada pela posição que o módulo ocupa na lista de módulos carregados (2 para as

[21] A interface *AdaptQoS* corresponde aos métodos públicos disponibilizados pela classe `LinuxQoSAdaptationManager`.

estratégias do serviço garantido e 5 para as estratégias de carga controlada). Assim, de acordo com a categoria informada numa requisição de substituição, o gerente pode remover o módulo já instalado na posição correspondente, substituindo-o pelo novo módulo.

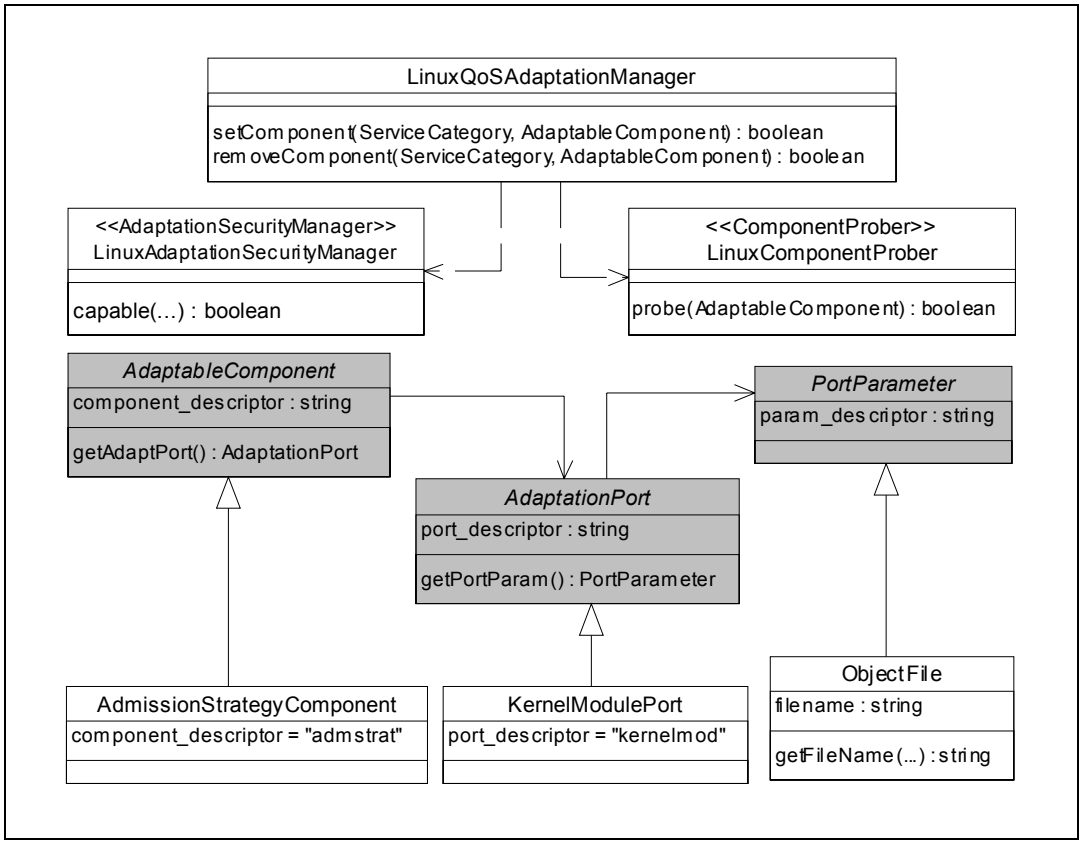

Figura 4.10 - Aplicação do framework para Adaptação de Serviços

4.4 Resumo do Capítulo

Neste Capítulo, foi descrito um exemplo real de aplicação da arquitetura QoSOS em um sistema operacional de propósito geral. Promover um GPOS a um cenário de uso da arquitetura trouxe a necessidade de se efetuar algumas modificações no código do *kernel* e da API de configuração do subsistema alvo da provisão. Deve-se ressaltar que esta não foi uma implementação eficiente, pois outros subsistemas devem participar da orquestração de recursos com o intuito de prover maior confiabilidade e predição sobre o comportamento do sistema como um todo. O objetivo de demonstrar a arquitetura, contudo, foi alcançado, pois o

ambiente desenvolvido permitiu a demonstração de como os frameworks da arquitetura podem ser aplicados em um cenário real.

Esse cenário foi propositadamente constituído como um complemento a um projeto anterior desenvolvido no Laboratório TeleMídia, ocasião em que não foram considerados os mecanismos de provisão de QoS nos sistemas operacionais agora implementados. O reaproveitamento de parte da interface de programação proposta por aquele trabalho possibilitou tanto a fácil integração entre os negociadores de diferentes níveis como o desenvolvimento de uma nova API genérica, estendida para a solicitação de serviços com QoS a sistemas operacionais.

5
Conclusões

Nesta dissertação, foram estudadas algumas das principais características que dificultam a provisão de QoS em sistemas operacionais de propósito geral, de forma a relacioná-las com soluções e novas abordagens propostas por grupos de pesquisa da área. Essa análise ficou restrita aos subsistemas de rede e de escalonamento de processos pela importância que representam para as aplicações multimídia distribuídas, constatada pelo grande número de trabalhos relacionados. A partir da discussão sobre as propostas de provisão de QoS, foi observado que os mecanismos utilizados em cada uma delas recorrem a aspectos comuns de implementação, passíveis de serem modelados de forma genérica. Por isso, o presente trabalho propôs uma arquitetura adaptável para a provisão de QoS nos subsistemas de rede e de escalonamento de processos de sistemas operacionais, independente de implementação, por meio da utilização de frameworks.

O emprego de frameworks como ferramenta de modelagem facilitou a identificação dos pontos de flexibilização (*hot-spots*) que permitem a instanciação da arquitetura para a implementação de vários cenários reais. Como conseqüência da característica adaptável da arquitetura QoSOS, foram incluídos mecanismos de criação e modificação de serviços acionados durante a operação do sistema, funcionalidade desejável dada a evolução contínua das aplicações e de suas necessidades.

Os exemplos de aplicação da arquitetura foram apresentados considerando a implementação de um subsistema de escalonamento de processos hierarquizado, regido por um meta-algoritmo capaz de emular o comportamento de diversos algoritmos de escalonamento. Por fim, foi descrita a implementação de um cenário real de uso, no qual o sistema operacional Linux sofreu pequenas alterações para oferecer uma infra-estrutura adaptável de suporte ao modelo Intserv sobre os buffers de comunicação de estações finais e roteadores.

5.1 Contribuições da dissertação

As principais contribuições do presente trabalho são as seguintes:

- Definição de um modelo de funcionamento de sistemas operacionais, considerando os subsistemas de rede e de escalonamento de processos.
- Proposta preliminar de extensão do meta-algoritmo LDS para o escalonamento hierárquico de recursos;
- Definição de uma arquitetura genérica adaptável de provisão de QoS em sistemas operacionais;
- Definição de uma API genérica de solicitação de serviços com QoS a sistemas operacionais;
- Colaboração em projeto internacional de código aberto para a construção de uma API de configuração dos mecanismos de controle de tráfego do Linux.

Definição de um modelo de funcionamento de sistemas operacionais

Concluindo a análise dos trabalhos relacionados, pôde-se construir um modelo simplificado do funcionamento de sistemas operacionais, considerando os subsistemas de rede e de escalonamento de processos. Tal modelagem foi capaz de ilustrar a dependência existente entre os subsistemas mencionados, e pôde servir como base para o modelo de orquestração de recursos da arquitetura definida.

Proposta preliminar de extensão do meta-algoritmo LDS

O estudo do meta-algoritmo LDS foi motivado pela capacidade de adaptação que ele poderia oferecer se implementado para fins reais de escalonamento e não de análise, como definido originalmente. Baseado no conceito de escalonamento hierárquico, foi proposta uma extensão do LDS que, além de torná-lo apto a implementar diversas estratégias de escalonamento simultaneamente, permitiria a utilização de outros algoritmos quando da impossibilidade de emulação. Deve-se salientar que essa proposta foi feita de

forma preliminar e encontra-se pendente com relação a análises de desempenho e testes de verificação.

Definição de uma arquitetura adaptável de provisão de QoS em sistemas operacionais

A definição da arquitetura QoSOS é a principal contribuição do presente trabalho. Embora o cenário utilizado para exemplificar o uso da arquitetura tenha sido o sistema operacional Linux, ela é suficientemente genérica para ser aplicada a vários cenários de uso. Além disso, o conceito de *hot-spots* específicos de serviço torna possível a modelagem de sistemas operacionais altamente adaptáveis. O framework de adaptação de serviços, ausente no conjunto original de frameworks genéricos para provisão de QoS, utiliza-se de mecanismos já presentes em alguns sistemas operacionais para possibilitar a modificação de estruturas internas do sistema em tempo de execução. Ele permite, assim, que novos serviços sejam criados e que políticas de provisão de QoS sejam modificadas sem a necessidade de interrupção de serviços ou atualizações de hardware nas estações e roteadores de uma rede.

Definição de API genérica de solicitação de serviços com QoS a sistemas operacionais

Para a implementação do cenário de uso apresentado no Capítulo 4 foi desenvolvida uma API de solicitação de serviços independente de plataforma e das políticas de provisão de QoS empregadas pelo sistema operacional. Como ressaltado, procurou-se manter essa interface semelhante, o quanto possível, à API definida por (Mota, 2001), já que os dois cenários se complementam. O trabalho supracitado propôs a implementação de negociadores Intserv, sem, entretanto, concretizar os mecanismos de reserva de recursos e de controle de admissão sobre os buffers de comunicação, mecanismos esses definidos no presente trabalho.

Colaboração em projeto internacional de código aberto

A implementação do cenário de uso da arquitetura também levou à utilização de uma API de nível mais baixo para a configuração de controle de tráfego no sistema Linux. A TCAPI foi a única ferramenta encontrada com esse objetivo, criada por um projeto de código aberto internacional gerenciado pela

IBM. Essa interface precisou ser estendida quando se percebeu que não suportava algumas das funcionalidades necessárias ao cenário. As várias modificações feitas pelo autor do presente trabalho foram submetidas ao administrador do projeto e todas foram aprovadas e integradas ao software original[22].

5.2 Trabalhos futuros

O presente trabalho abriu possibilidades de aprofundamento nos seguintes assuntos, passíveis de exploração em trabalhos futuros:

- Inclusão de outros recursos relevantes do sistema no modelo de orquestração;
- Análise e implementação da proposta de extensão do algoritmo LDS;
- Estudo de formas de verificação dos aspectos de segurança e da consistência nos mecanismos de adaptação;
- Exploração dos mecanismos de adaptação por outras funções de sistemas operacionais;
- Aperfeiçoamento do protótipo de implementação da arquitetura sobre o sistema operacional Linux;
- Estudo para portar o protótipo desenvolvido para utilização em set-top boxes de sistemas de TV interativa.

Inclusão de outros recursos relevantes do sistema no modelo de orquestração

O estudo de outros subsistemas relevantes para a provisão de QoS em sistemas operacionais poderia trazer contribuições e refinamentos aos mecanismos definidos pela arquitetura. Se recursos como memória principal, sistema de paginação e acesso a disco forem incluídos no modelo de orquestração, o comportamento do sistema poderá ser predito com maior exatidão, oferecendo maior confiabilidade aos serviços instanciados a partir da arquitetura. Uma boa metodologia para fundamentar o modelo de orquestração seria descrever o

[22] Disponível em http://www-124.ibm.com/developerworks/projects/tcapi/.

relacionamento entre todos os recursos por meio de uma modelagem geral do funcionamento do sistema, como foi feito entre CPU e buffers de comunicação na Seção 2.3.

Análise e implementação da proposta de extensão do algoritmo LDS

A infra-estrutura de escalonamento de processos baseada no meta-algoritmo LDS hierárquico foi aqui abordada como uma proposta preliminar e utilizada para fins ilustrativos. Um trabalho que confirme a viabilidade de seu uso e promova sua implementação em um sistema operacional constituirá uma grande contribuição por disponibilizar uma porta de adaptação para estratégias de escalonamento de CPU, cujos componentes não necessitam ser codificados em linguagem de programação.

Estudo de formas de verificação de segurança e consistência sobre a adaptação

A aplicação de diretivas de segurança sobre certos mecanismos de provisão tem sido pouco explorada nos trabalhos sobre QoS. As solicitações de reservas de recursos devem ser disponibilizadas apenas a usuários autorizados e o gerenciamento de adaptabilidade do sistema deve ser cercado de cuidados. As formas de verificação dos aspectos de segurança levantados no Capítulo 3 devem ser estudadas de modo a solucionar questões do tipo: “Como verificar a confiabilidade de um componente?”; “Como assegurar a integridade do sistema frente à introdução de um componente?”; “Como verificar se um componente contém ações mal intencionadas?”; “Como monitorar as portas de adaptação para que não sejam usadas para outros fins, como invasão?”. Existe, ainda, a necessidade de verificação da consistência do sistema diante de algumas operações, como a remoção e a substituição de componentes. Uma análise detalhada sobre métodos capazes de manter uma base de informações sobre dependências entre componentes seria, portanto, de grande interesse.

Exploração dos mecanismos de adaptação para outras funções do sistema operacional

A aplicação do modelo de adaptabilidade em outras funções do sistema operacional é desejável, pois permitiria a configuração desses serviços por parte das aplicações ou de protocolos de reconfiguração em tempo de execução. Um

exemplo é a especificação de componentes que implementam tarefas específicas de uma pilha de protocolos adaptável, que é uma funcionalidade de baixo nível muito útil para ferramentas de programação de aplicações distribuídas, como a proposta por (Schmidt, 1994).

Aperfeiçoamento do protótipo de implementação da arquitetura sobre o sistema operacional Linux

A implementação de um protótipo completo e eficaz para a provisão de QoS em um GPOS específico vem a ser a validação prática de todas as estruturas apresentadas pela arquitetura proposta, principalmente o modelo de orquestração de recursos. O protótipo apresentado no Capítulo 4 pode ser estendido para que o código implementado no nível de usuário seja transportado para o *kernel* do sistema Linux. O próprio subsistema de escalonamento de pacotes (LinuxTC) pode ser aperfeiçoado para prover adaptabilidade na introdução de novas estratégias de escalonamento, utilizando também o algoritmo LDS estendido. Outro complemento do trabalho é a instanciação da arquitetura sobre o subsistema de escalonamento de processos, para o gerenciamento de QoS na CPU.

Estudo para portar o protótipo desenvolvido para utilização em set-top boxes de sistemas de TV interativa

Sistemas de TV interativa são aplicações multimídia distribuídas caracterizadas pelos requisitos já descritos no presente trabalho. Os terminais de recepção do sistema são chamados de set-top boxes (O'Driscoll, 2000), computadores de pequeno porte capazes de manipular os fluxos de áudio e vídeo para exibição e interação com o usuário. A escassez de recursos nesses terminais traz a necessidade de um estudo para a limitação das funcionalidades do sistema operacional com QoS àquelas estritamente necessárias. Além disso, os set-top boxes devem ser alvo de pesquisa para a identificação de outras necessidades de gerenciamento de QoS, por possuírem dispositivos de entrada e saída incomuns na maioria dos sistemas computacionais.

6
Referências Bibliográficas

ALMESBERGER, Werner. **Linux network traffic control**: Implementation overview. Implementation details, 1999.
Disponível em ftp://icaftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz.

BARABANOV, Michael. **A linux-based real-time operating system**. Master Degree dissertation, New Mexico Institute of Mining Technology, 1997.
Disponível em ftp://ftp.rtlinux.com/pub/rtlinux/papers/thesis.ps.

BARRIA, Marta. **Algoritmos para QoS em redes de computadores**. Tese de Doutorado, Pontifícia Universidade Católica do Rio de Janeiro, 2001.

BERNET, Yoram et al. **Winsock generic QoS mapping**. Windows Networking Group Draft, 1998.
Disponível em ftp://ftp.microsoft.com/bussys/wINSOCK/winsock2/gqos_spec.doc.

BLACK, Richard, et al. **Protocol implementation in a vertically structured operating system**. Proceedings of 22nd Annual Conference on Local Computer Networks (LCN'97), 1997.
Disponível em http://www.cl.cam.ac.uk/Research/SRG/netos/old-projects/pegasus/papers/lcn-9704.ps.gz.

BLAKE, S. et al. **An architecture for differentiated services**. IETF Request for Comments (RFC2475), 1998.
Disponível em http://www.ietf.org/rfc/rfc2475.txt.

BRADEN, R.; CLARK, D.; SHENKER, S. **Integrated services in the internet architecture**: an overview. IETF Request for Comments (RFC1633), 1994.
Disponível em http://www.ietf.org/rfc/rfc1633.txt.

BRADEN, R., BERSON, S., HERZOG, S., JAMIN, S. E ZHANG, L. **Resource reservation protocol (RSVP)**: Version 1 Functional Specification. IETF Request for Comments (RFC2205), 1997.
Disponível em http://www.ietf.org/rfc/rfc2205.txt.

CAMPBELL, Andrew. **A quality of service architecture**. PhD thesis, Computing Department, Lancaster University, 1996.
Disponível em http://comet.columbia.edu/~campbell/andrew/publications/papers/thesis.pdf.

COLCHER, Sérgio; GOMES, A. Tadeu; SOARES, L. Fernando. **Um meta modelo para a engenharia de serviços de telecomunicações**. XVIII Simpósio Brasileiro de Redes de Computadores (SBRC'2000), 2000. Disponível em ftp://ftp.telemidia.puc-rio.br/pub/docs/conferencepapers/2000_05_colcher.ps.gz.

COULSON, Geoff; BLAIR, Gordon. **Architectural principles and techniques for distributed multimedia application support in operating systems**. ACM Operating Systems Review, v.29, n.4, p.17-24, 1995. Disponível em ftp://ftp.comp.lancs.ac.uk/pub/mpg/MPG-95-09.ps.

DEMERS, A.; KESHAV, S.; SHENKER, S. **Analysis and simulation of a fair queueing algorithm**. Journal of Internetworking: Research andExperience, vol. 1, pp 3-26, 1990.

DRUSCHEL, Peter; BANGA, Gaurav. **Lazy receiver processing (LRP)**: a network subsystem architecture for server systems. Proceedings of 2nd Symposium on Operating System Design and Implementation (OSDI'96), p. 261-275, 1996. Disponível em http://www.cs.rice.edu/CS/Systems/LRP/osdi96.ps.

FLOYD, Sally; JACOBSON, Van. **Link-sharing and resource management models for packet networks**. IEEE/ACM Transactions on Networking, Vol. 3 No. 4, pp. 365-386, 1995 Disponível em http://www.icir.org/floyd/papers/link.pdf.

FORD, Bryan; SUSARLA, Sai. **CPU inheritance scheduling**. Proceedings of 2nd Symposium on Operating Systems Design and Implementation (OSDI'96), 1996. Disponível em http://www.cs.utah.edu/flux/papers/inherit-sched.ps.gz.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns**: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

GOMES, Antônio T. **Um framework para provisão de QoS em ambientes genéricos de processamento e comunicação**. Dissertação de Mestrado, Pontifícia Universidade Católica do Rio de Janeiro, 1999. Disponível em ftp://ftp.telemidia.puc-rio.br/pub/docs/theses/1999_05_gomes.pdf.

GOPALAKRISHNA, Raman; PARULKAR, Guru. **Efficient quality of service support in multimedia computer operating systems**. Technical Report WUCS-TM-94-04, Washington University, 1994. Disponível em http://www.arl.wustl.edu/arl/refpapers/gopal/wucs-94-26.ps.

GOVINDAN, Ramesh; ANDERSON, David. **Scheduling and IPC mechanisms for continuous media**. Proceedings of 13th ACM Symposium on Operating System Principles, 1991. Disponível em ftp://ftp.isi.edu/pub/govindan/os_mechanisms.ps.

GOYAL, Pawan; VIN, Harrick; CHENG, Haichen. **Start-time fair queuing**: a scheduling algorithm for integrated services packet switching networks. Proceedings of ACM SIGCOMM'96, p. 157-168, 1996. Disponível em http://www.cs.utexas.edu/users/dmcl/papers/ps/SIGCOMM96.ps.

GOYAL, Pawan; GUO, Xingang; VIN, Harrick. **A hierarchical CPU scheduler for multimedia operating systems**. Proceedings of 2nd Symposium on Operating System Design and Implementation (OSDI'96), p. 107-122, 1996. Disponível em http://www.cs.utexas.edu/users/dmcl/papers/ps/OSDI96.ps.

LAKSHMAN, K.; YAVAKTAR, Raj; FINKEL, Raphael. **Integrated CPU and network-I/O QoS management in an endsystem**. Proceedings of 5th International Workshop on Quality of Service (IWQOS'97), Columbia University, p.167-178, 1997. Disponível em http://www.ccs.uky.edu/~lakshman/papers/iwqos97.ps.

LAZAR, A. **Programming telecommunication networks**, IEEE Network Magazine, 1997.

LEE, Chen; et. al. **Predictable communication protocol processing in real-time mach**. Proceedings of the Real Time Technology and Applications Symposium, 1996. Disponível em http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/rtas96.ps.

LESLIE, Ian et al. **The design and implementation of an operating system to support distributed multimedia applications**. IEEE Journal of Selected Areas in Communications, 1996. Disponível em http://www.cl.cam.ac.uk/Research/SRG/netos/old-projects/pegasus/papers/jsac-jun97.ps.gz.

LIU, C.; LAYLAND, James. **Scheduling algorithms for multiprogramming in a hard-real-time environment**. Journal of the Association for Computing Machinery, v. 20, p. 46-61, 1973. Disponível em http://www.acm.org/pubs/articles/journals/jacm/1973-20-1/p46-liu/p46-liu.pdf.

MAXWELL, Scott. **Kernel do linux**. Makron Books, São Paulo, 2000.

MEHRA, Ashish; INDIRESAN, Atri; SHIN, Kang. **Structuring communication software for quality of service guarantees**. Proceedings of 17th Real Time Systems Symposium, 1996. Disponível em http://kabru.eecs.umich.edu/papers/publications/1997/ashish_tse97.ps.gz.

MERCER, Clifford; ZELENICA. Jim; RAJKUMAR, R. **On predictable operating system protocol processing**. Technical Report CMU-CS94-165, Carnegie Mellow University, 1994.
Disponível em http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/cmu-cs-94-165.ps.

MICROSOFT Corp. **Microsoft windows nt server networking guide**. Microsoft Windows NY Resource Kit, Microsoft Press, 1996.

MOTA, Oscar Thyago. **Uma arquitetura adaptável para provisão de QoS na internet**. Dissertação de Mestrado, Pontifícia Universidade Católica do Rio de Janeiro, 2001.
Disponível em ftp://ftp.telemidia.puc-rio.br/pub/docs/theses/2001_05_mota.zip. O'DRISCOLL, Gerard. **The essential guide to digital set-top boxes and interactive TV**. Premtice Hall, 2000.

OLSHEFSKI, David. **Notes on linux network QoS**: TCAPI version 1.0. Work in progress, 2001.
Disponível em ftp://www-126.ibm.com/pub/tcapi/tcapi.tar.gz.

OPARAH, Don. **Adaptive resource management in a multimedia operating system**. Proceedings of 8th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 98), 1998
Disponível em http://www.cl.cam.ac.uk/Research/SRG/nossdav98/papers/nossdav98-049.ps.gz.

PREE, W. **Design patterns for object-oriented software development**. Addison Wesley, 1995.

RADHAKRISHNAN, S. **Linux advanced network overview**: implementation details, 1999.
Disponível em http://qos.ittc.ukans.edu/howto.pdf.

REGEHR, John. **Using hierarchical scheduling to support soft real-time applications in general-purpose operating systems**. PhD thesis, University of Virginia, 2001.
Disponível em http://www.cs.utah.edu/~regehr/papers/diss/regehr-diss-single.pdf.

RUSLING, David. **The linux kernel**. Free digital book, 1996.
Disponível em http://rainbow.mimuw.edu.pl/SO/Linux-doc/Linux-Kernel.ps.gz.

SCHMIDT, Douglas. **The adaptive communication environment**: an object-oriented network programming toolkit. Proceedings of 12th Sun Users Group Conference, 1994.
Disponível em http://www.cs.wustl.edu/~schmidt/PDF/SUG-94.pdf.

SHENKER, S.; PARTRIDGE C.; GUERIN R. **Specification of guaranteed quality of service**. IETF Request for Comments (RFC2212), 1997.
Disponível em http://www.ietf.org/rfc/rfc2212.txt.

SHENKER, S.; WROCLAWSKI, J. General **Characterization parameters for integrated service network elements**. IETF Request for Comments (RFC2215), 1997.
Disponível em http://www.ietf.org/rfc/rfc2215.txt.

SHI, Sherlia; PARULKAR, G.; GOPALAKRISHNAN, R. **A TCP/IP implementation with endsystem QoS**. Technical Report WUCS-98-10, Washington University, 1998.
Disponível em http://www.cs.wustl.edu/cs/techreports/1998/wucs-98-10.ps.Z.

SOARES, Luiz Fernando. **Modelagem e simulação discreta de sistemas**. VII Escola de Computação, IME-USP, 1990.

TANENBAUM, Andrew. **Modern operation systems**. Prentice Hall, 1992.

RATIONAL SOFTWARE CORPORATION. **Unified modeling language**: notation guide, 1997.

WALTON, Sean. **Linux threads frequently asked questions (FAQ)**, 1996.
Disponível em http://linas.org/linux/threads-faq.html.

WRIGHT, Gary; STEVENS, W. **TCP/IP illustrated, volume 2**: the implementation. Addison-Wesley, 1995.