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.

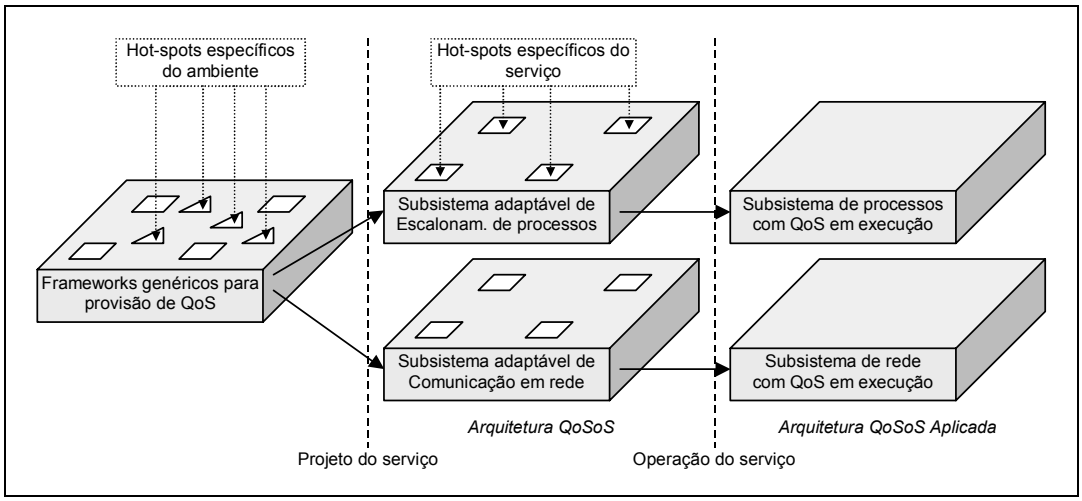

Figura 3.1 -Tipos de *hot-spots* de uma arquitetura modelada pelos frameworks genéricos de (Gomes, 1999)

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.

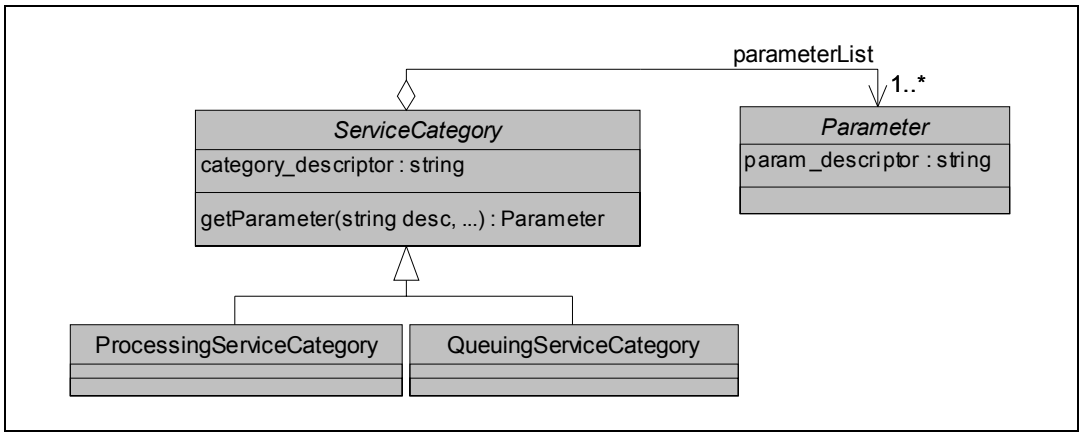

Figura 3.2 - Framework para Parametrização de Serviços

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.

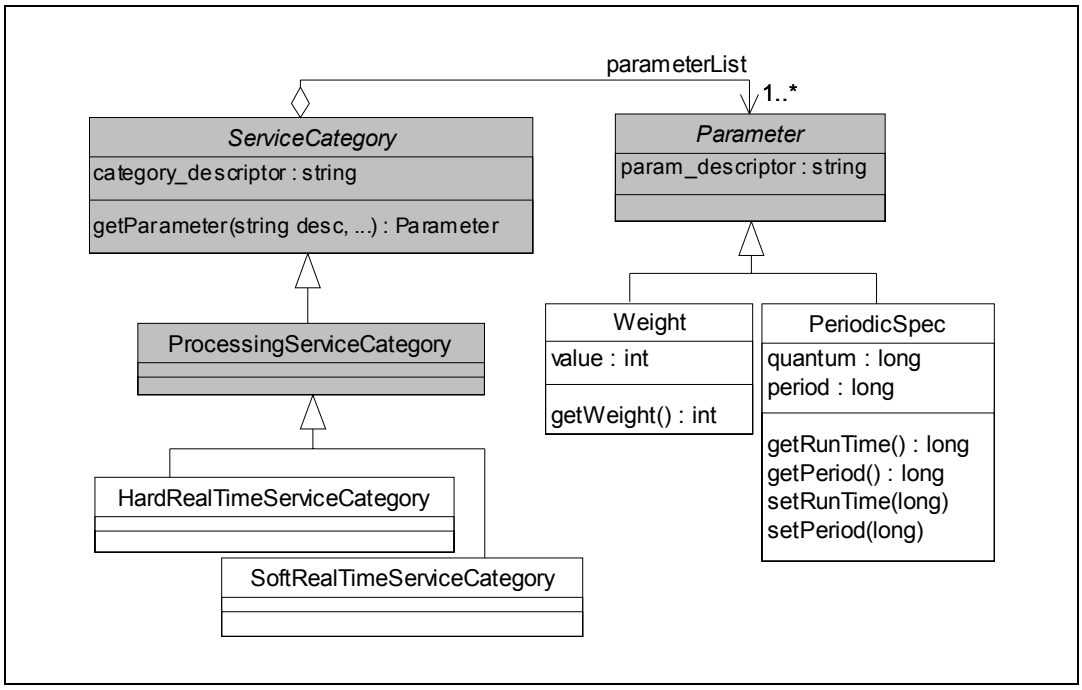

Figura 3.3 - Exemplo de aplicação do Framework para Parametrização de Serviços

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.

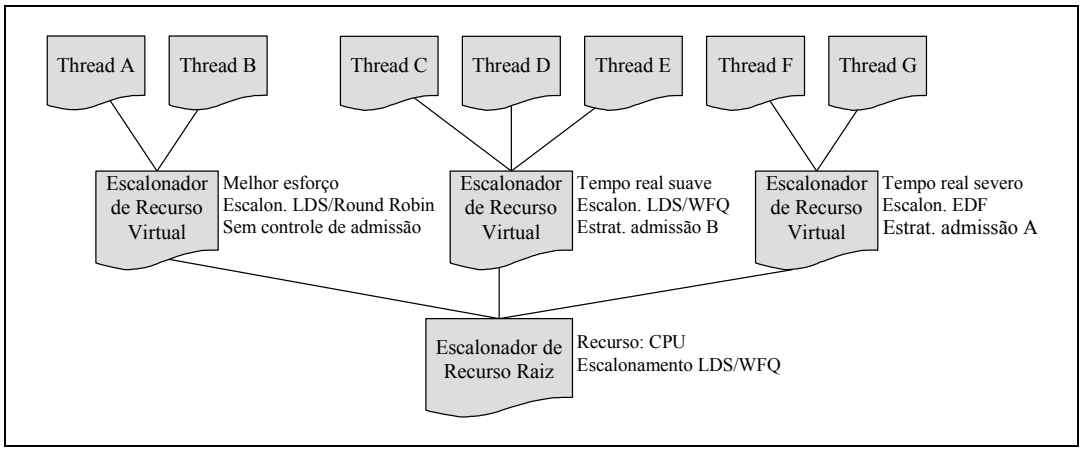

Figura 3.4 - Exemplo de árvore de recursos virtuais para o recurso CPU

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()`).

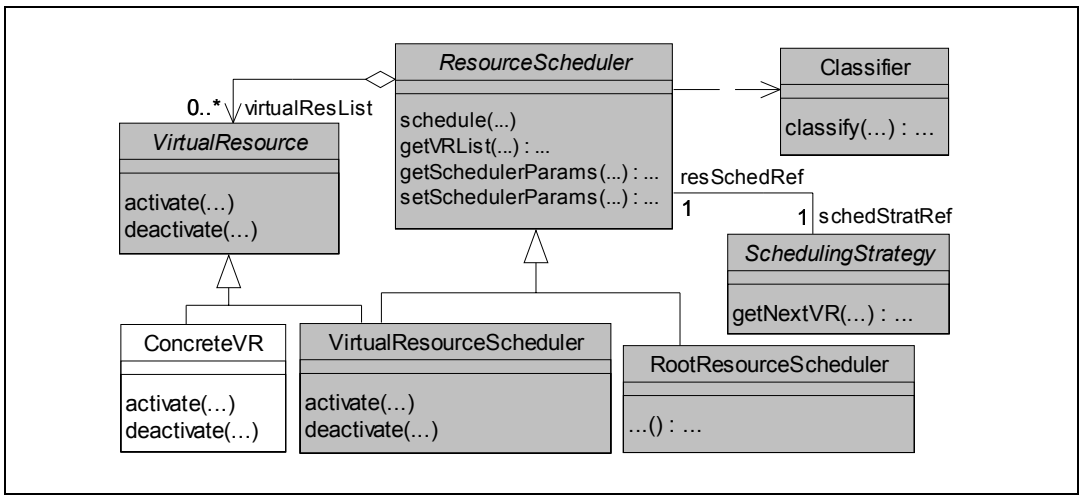

Figura 3.5 - Framework para Escalonamento de Recursos

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.

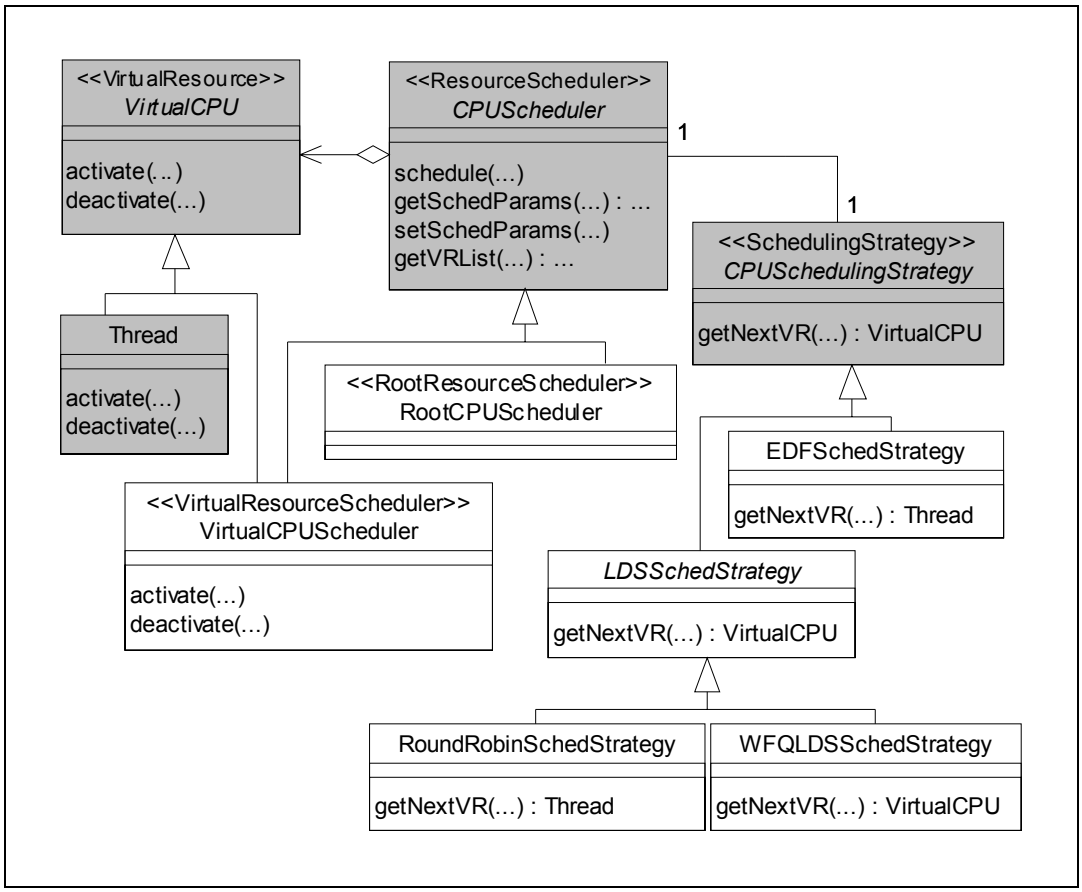

Figura 3.6 - Exemplo de aplicação do Framework para Escalonamento de Recursos

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.

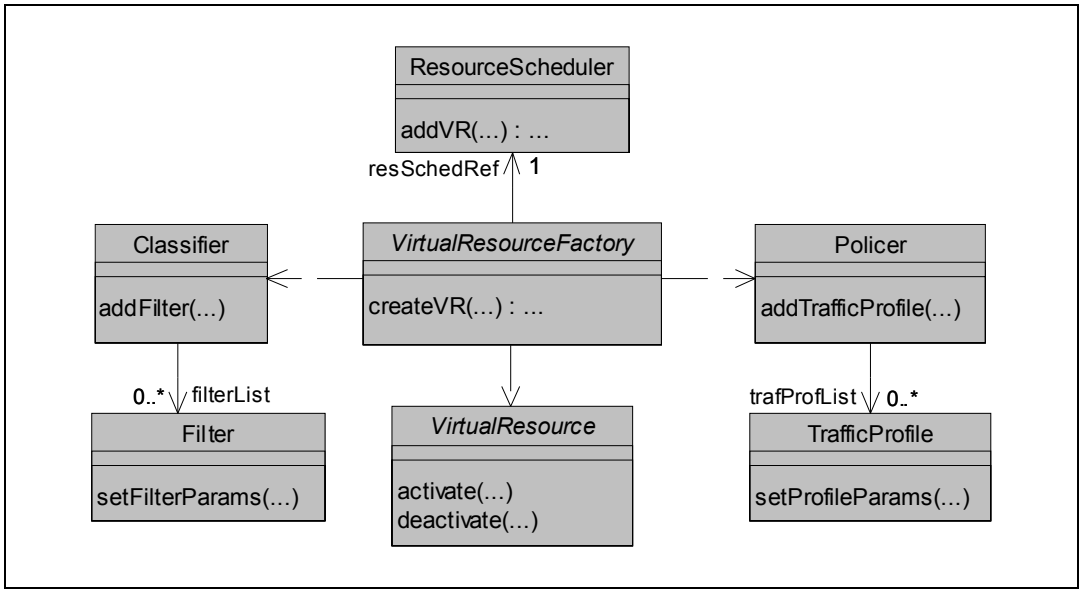

Figura 3.7 - Framework para Alocação de Recursos

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.

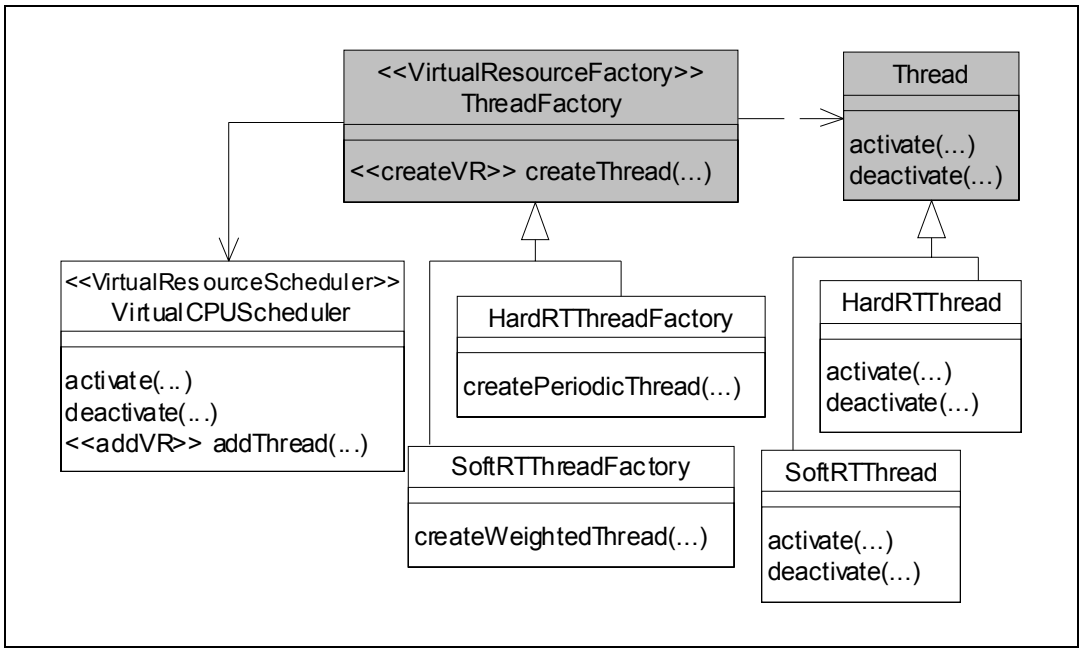

Figura 3.8 - Exemplo de aplicação do Framework para Alocação de Recursos

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.

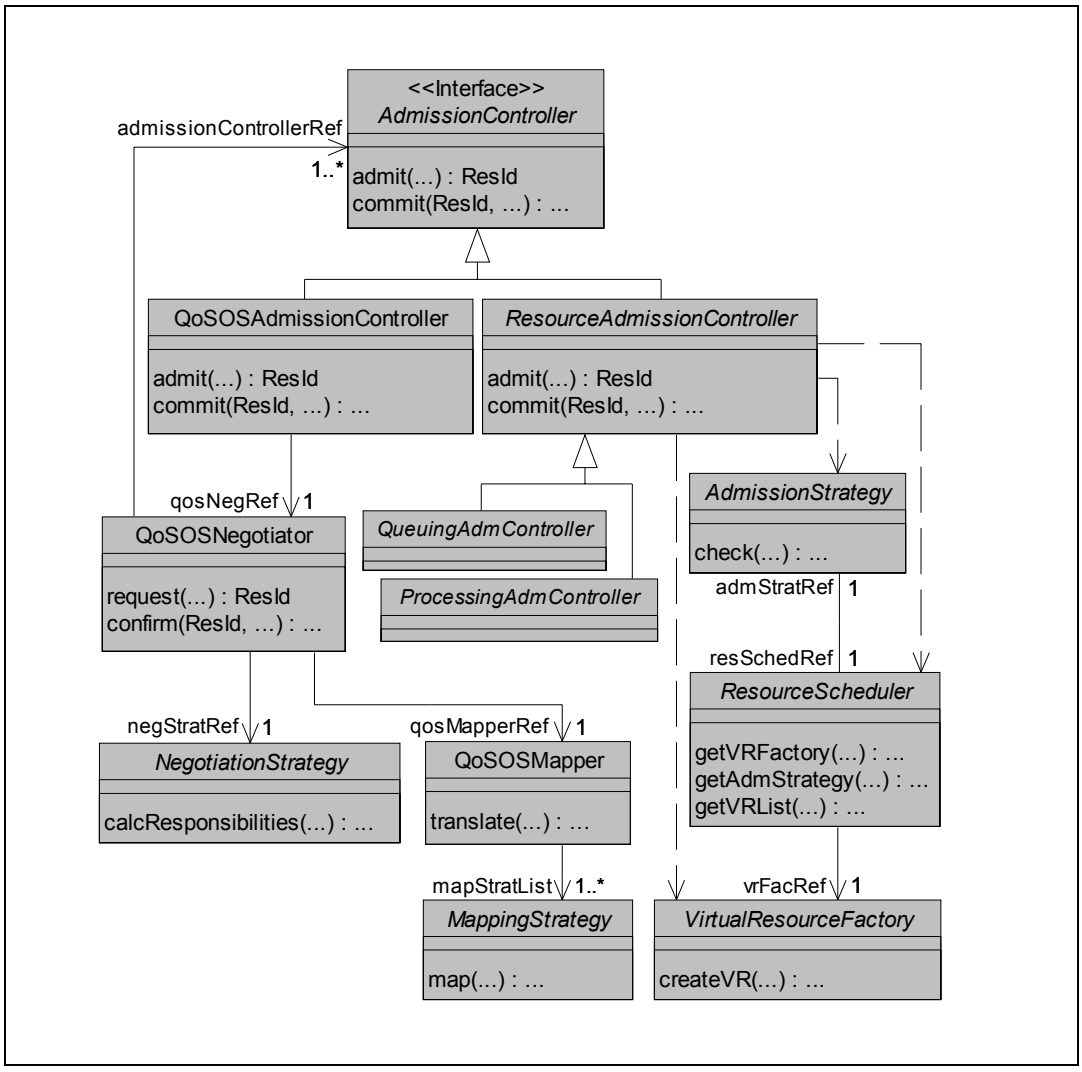

Figura 3.9 - Framework para Negociação de QoS

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.

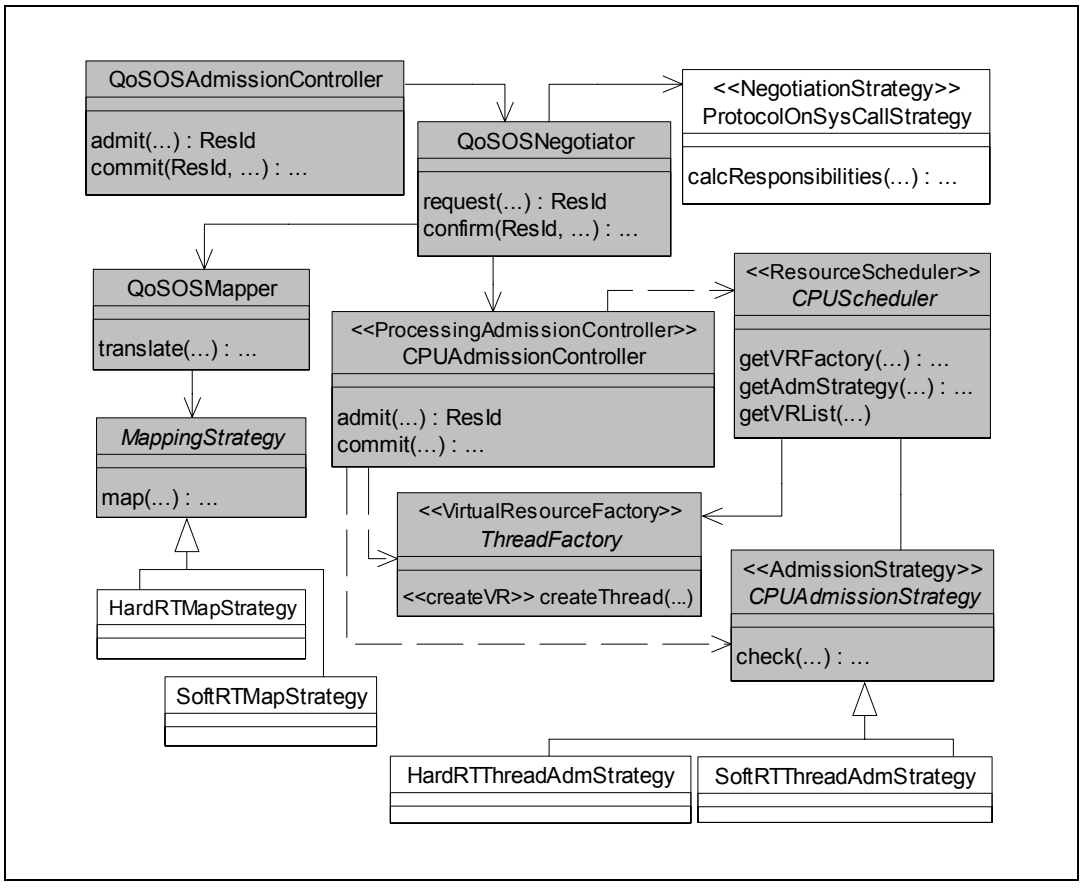

Figura 3.10 - Exemplo de aplicação do Framework para Negociação de QoS

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.

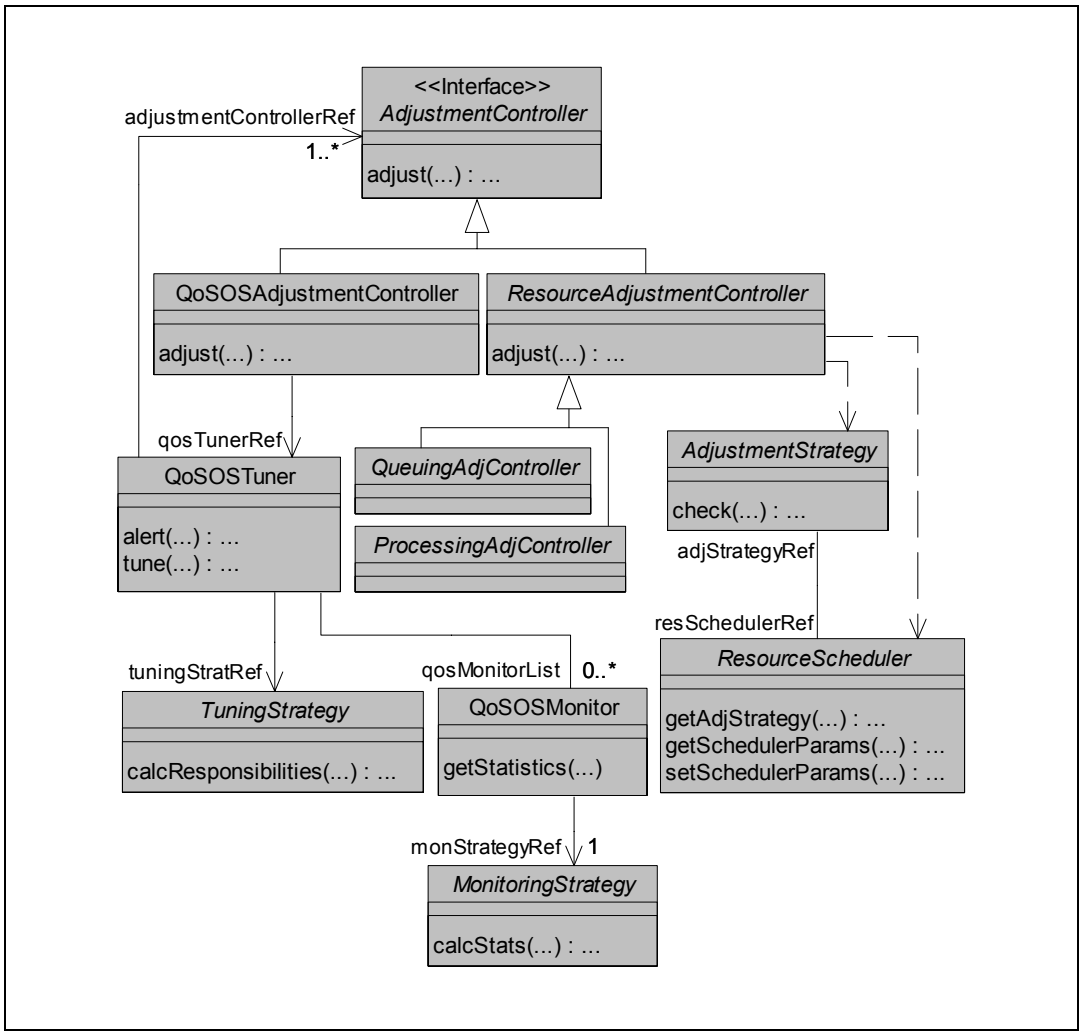

Figura 3.11 - Framework para Sintonização de QoS

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()`.

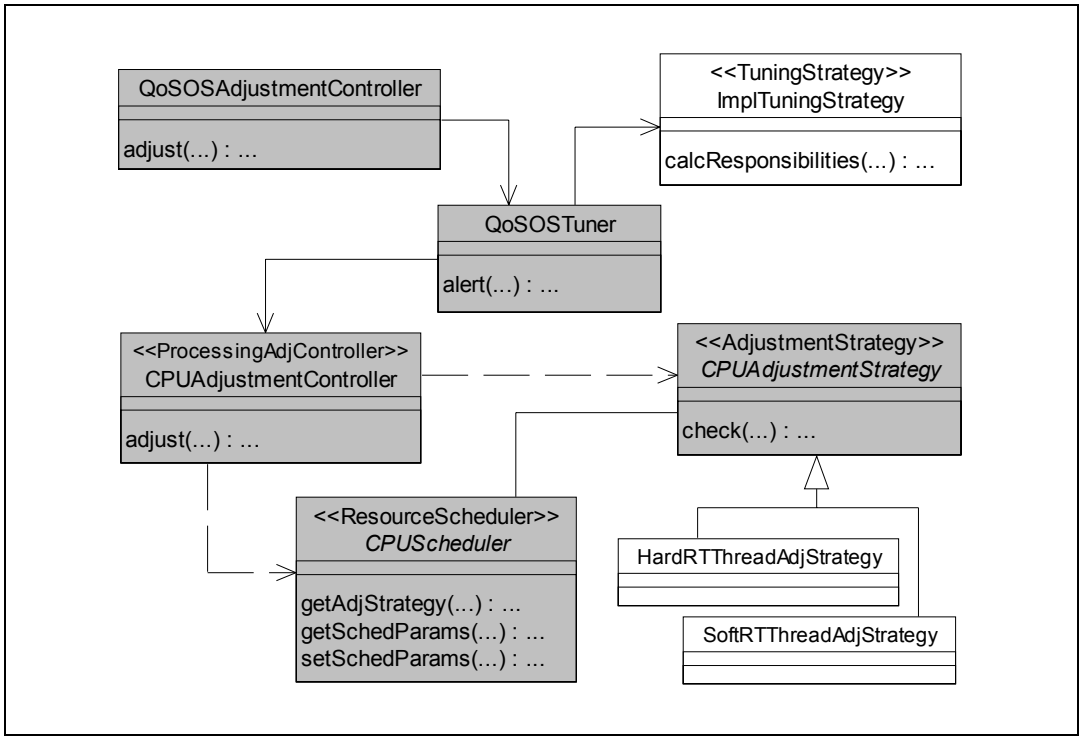

Figura 3.12 - Exemplo de aplicação do Framework para Sintonização de QoS

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.

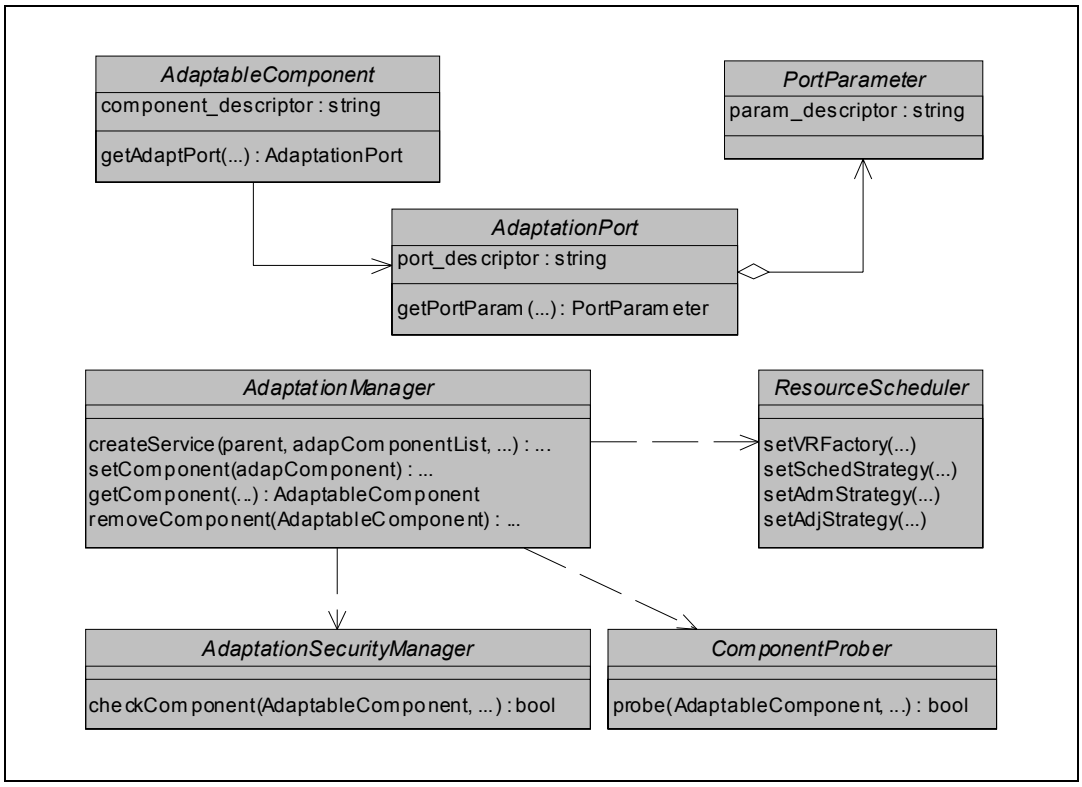

Figura 3.13 - Framework para Adaptação de Serviços

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.

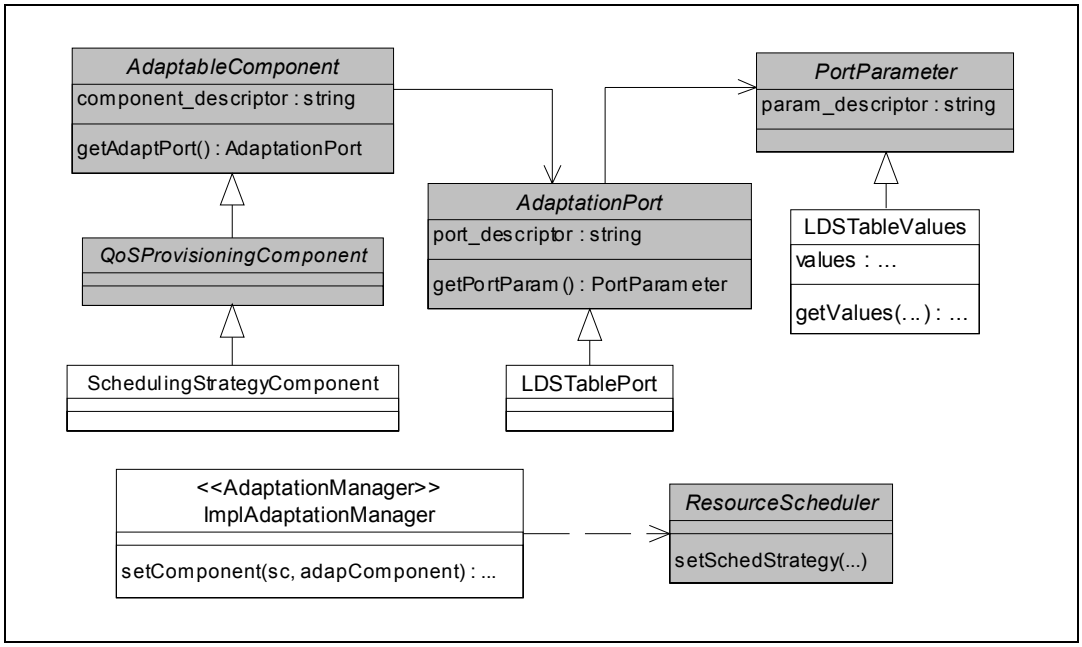

Figura 3.14 - Exemplo de aplicação do Framework para Adaptação de Serviços

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.