Universidade Federal da Bahia
Instituto de Matemática

Bacharelado em Ciência da Computação

ESPECIFICAÇÃO E IMPLEMENTAÇÃO DE UM MECANISMO DE MONITORAMENTO PARA UM PROVEDOR DE QUALIDADE DE SERVIÇO DISTRIBUÍDO

Hugo Vinicius Vaz Braga

TRABALHO DE GRADUAÇÃO

Salvador
16 de Dezembro de 2008

HUGO VINICIUS VAZ BRAGA

ESPECIFICAÇÃO E IMPLEMENTAÇÃO DE UM MECANISMO DE MONITORAMENTO PARA UM PROVEDOR DE QUALIDADE DE SERVIÇO DISTRIBUÍDO

Esta Trabalho de Graduação foi apresentada ao Bacharelado em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Sérgio Gorender

Salvador
16 de Dezembro de 2008

TERMO DE APROVAÇÃO

HUGO VINICIUS VAZ BRAGA

ESPECIFICAÇÃO E IMPLEMENTAÇÃO DE UM MECANISMO DE MONITORAMENTO PARA UM PROVEDOR DE QUALIDADE DE SERVIÇO DISTRIBUÍDO

Esta Trabalho de Graduação foi julgada adequada à obtenção do título de Bacharel em Ciência da Computação e aprovada em sua forma final pelo Bacharelado em Ciência da Computação da Universidade Federal da Bahia.

Salvador, 16 de Dezembro de 2008

Prof. Sérgio Gorender (Orientador)
Universidade Federal da Bahia

Prof. Flávio Morais de Assis Silva
Universidade Federal da Bahia

Prof. Raimundo José de Araújo Macêdo
Universidade Federal da Bahia

RESUMO

A capacidade de se adaptar dinamicamente às condições distintas de execução é uma questão muito importante quando se trata de sistemas distribuídos cuja Qualidade de Serviço (*Quality of Service* - QoS) negociada nem sempre pode ser entregue entre os processos (GORENDER; MACêDO; RAYNAL, 2007). Dado esta motivação, (GORENDER; MACêDO; RAYNAL, 2007) propuseram um modelo adaptativo de programação para sistemas distribuídos tolerantes à falhas. Para que este modelo possa funcionar corretamente, ele necessita obter informações sobre a QoS (além da execução de alguns outros serviços) provida aos canais de comunicação. Estas informações são obtidas através da interface padronizada denominada *QoS Provider* (QoSP). Além da padronização, o *QoS Provider* visa encapsular todos os detalhes a respeito das arquiteturas de QoS que estão sendo utilizadas. O QoSP pode ser resumido em dois grandes serviços: negociação e monitoração de QoS. Este trabalho visa desenvolver, especificar e implementar o mecanismo de monitoramento do QoSP. Este mecanismo corresponde ao serviço de monitoramento mencionado anteriormente. Este mecanismo engloba três funcionalidades: monitorar a QoS que está sendo provida a um canal de comunicação, verificar se há tráfego em um canal durante um intervalo de tempo e realizar um monitoramento automático.

Palavras-chave: Sistemas distribuídos, QoS, monitoramento.

ABSTRACT

The capability of dynamically adapting to distinct runtime conditions is an important issue when designing distributed systems where negotiated quality of service (QoS) cannot always be delivered between processes (GORENDER; MACêDO; RAYNAL, 2007). Given this motivation, (GORENDER; MACêDO; RAYNAL, 2007) proposed an adaptive programming model for fault-tolerant distributed systems. For this model to function properly it needs to obtain information about the QoS (besides the execution of some other services) provided to the communication channels. This information is obtained through the standardized interface called *QoS Provider* (QoSP). In addition to standardization, the *QoS Provider* aims to encapsulate all the details about the underlying QoS architectures. The QoSP can be summarized in two major services: admission and QoS monitoring. This work aims to develop, implement and specify the QoS monitoring mechanism. This mechanism is the QoS monitoring service mentioned earlier. This mechanism comprises three functions: obtain the current QoS of a channel, assess if message flow happens in a channel within a period of time and carry out a periodic monitoring.

Keywords: Distributed systems, Quality of Service (QoS), monitoring.

SUMÁRIO

LISTA DE FIGURAS

Capítulo 1

INTRODUÇÃO

De acordo com (GORENDER; MACêDO; RAYNAL, 2007), a capacidade de se adaptar dinamicamente às condições distintas de execução é uma questão muito importante quando se trata de sistemas distribuídos cuja Qualidade de Serviço (*Quality of Service* - QoS) negociada nem sempre pode ser entregue entre os processos. Sistemas distribuídos são definidos como aqueles nos quais os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas trocando mensagens (COULOURIS; DOLLIMORE; KINDBERG, 2007). Com relação à degradação da QoS, (JIANG; THAM; KO, 2000) afirma que esta é quase sempre inevitável e que a reserva de recursos não é suficiente para que a QoS seja sustentada. Levando em consideração este contexto, (GORENDER; MACêDO; RAYNAL, 2007) propuseram um modelo adaptativo de programação para sistemas distribuídos tolerantes à falhas.

Este modelo (também conhecido como modelo HA - *Hybrid and Adaptative*) é considerado híbrido, visto que é composto por partes assíncronas (nas quais não existem limites temporais para processamento e transferência de mensagens) e síncronas (nas quais existem limites temporais) mantidas por QoS, e adaptativo, visto que a QoS se altera e o modelo necessita se adaptar às condições de QoS. Para que o modelo HA possa funcionar corretamente, ele necessita obter informações sobre a QoS que está sendo provida a cada canal de comunicação.

Tais informações são providas através do dispositivo de software denominado QoS Provider (QoSP). O QoSP corresponde a uma interface padronizada que cria e gerencia canais de comunicação com QoS. O QoSP não é uma arquitetura de QoS, mas sim um invólucro para um conjunto de arquiteturas para prover QoS. Dessa maneira, mesmo que o ambiente de comunicação seja alterado (novas arquiteturas de QoS sejam utilizadas), os processos que usufruem desta interface não sofrerão mudança alguma, visto que a interface do QoSP não muda. O QoSP deve se comunicar com as as arquiteturas de QoS existentes no ambiente para obter informações relativas a QoS de um canal.

A interface do QoSP é definida através de cinco funções: criação de canal, cálculo de *delay*, negociação de QoS, monitoração de QoS e verificação de canal. Apesar das cinco funções, o QoSP pode ser resumido em dois grandes serviços: negociação e monitoração

de QoS. O serviço de monitoração engloba as funções de monitoração de QoS e de verificação de canal. A função de monitoração de QoS retorna a QoS que está sendo provida a um canal, enquanto que a função de verificação além de desempenhar tal funcionalidade ela verifica se há tráfego em um canal durante um determinado intervalo de tempo (estas funções serão detalhadas no capítulo 3). Uma outra funcionalidade do serviço de monitoração é o monitoramento automático, que corresponde a um monitoramento que é executado mesmo quando não há uma solicitação explícita ao QoSP.

A monitoração de QoS é um mecanismo utilizado por outros componentes para obter um *feedback* sobre o estado atual da QoS, para que medidas possam ser tomadas baseado neste *feedback*. Isto se aplica bem ao modelo proposto HA, visto que para que o modelo HA possa afirmar que um determinado processo falhou, ele consultar o QoSP (mas especificadamente, o mecanismo de monitoramento). De acordo com (AURRECOECHEA; CAMPBELL; HAUW, 1996), mecanismos de monitoração de QoS são partes fundamentais de arquiteturas de QoS. O módulo (estou utilizando este termo intercambiável com mecanismo) de monitoração do QoSP não é um componente de arquitetura de QoS, mas se comporta como um para o QoSP, visto que é ele quem retorna ao QoS que está sendo provida ao longo dos componentes de software e hardware de um canal (AURRECOECHEA; CAMPBELL; HAUW, 1996). Logo justifica-se a adoção de princípios e de um modelo de sistema de monitoramento.

Este trabalho visa desenvolver, especificar e implementar o mecanismo de monitoramento do QoS Provider, sendo que este último tem como base a arquitetura Diffserv e conta com um sistema operacional de tempo real instalado nos *hosts*. O desenvolvimento deve ser baseado em um modelo e em princípios de sistema de monitoramento, além da utilização de protocolos que diminuam o *overhead* da rede. A utilização de princípios assim como de protocolos que diminuam o *overhead* visa não tornar o sistema de monitoramento um gargalo, mas sim um componente que auxilie a tomada de decisões. O desenvolvimento se dará através da especificação dos algoritmos que compõem o mecanismo de monitoramento assim como de um protocolo utilizado pelo mesmo. A especificação do sistema se dará através da modelagem do mesmo utilizando diagramas UML. Ao final da implementação, o sistema deverá ser testado.

Capítulo 2

QOS

QoS é o acrônimo para *Quality of Service*, sendo um termo amplamente utilizado na literatura, mas em poucos artigos ele é definido. De acordo com (GODERIS, 2001), QoS se refere a um serviço sendo oferecido onde um ou mais parâmetros de performance (isto é, vazão, atraso, perda e *jitter*) podem ser quantificados. (WANG, 2001) define QoS como sendo "a capacidade de prover garantias de recursos e diferenciação de serviços em uma rede". Quando um canal de comunicação possui recursos reservados e algum nível de prioridade com relação a outros canais, podemos dizer que este canal possui uma QoS.

2.1 ARQUITETURAS DE QOS

Diversas arquiteturas para prover QoS têm sido desenvolvidas nos últimos anos (AURRECOECHEA; CAMPBELL; HAUW, 1996). A motivação para a utilização de mecanismos de QoS, consequentemente para o desenvolvimento de arquiteturas de QoS, foi a necessidade de prover priorização e proteção a alguns fluxos de tráfegos (EL-GENDY; BOSE; SHIN, 2003), mas especificadamente aos tráfegos provindos das aplicações de tempo real, visto que estas necessitam de garantias temporais (previsibilidade). Este conjunto inclui os princípios que regem a construção de uma arquitetura genérica, a especificação da QoS que é necessária para capturar os requisitos da aplicação, e por último os mecanismos pelos quais os requisitos da aplicação serão atendidos. Basicamente os mecanismos podem ser divididos em dois tipos: provisão e controle/gerenciamento de QoS. Enquanto que o primeiro visa a reserva dos recursos, o outro visa a manutenção da QoS. Um dos componentes necessários para que esta manutenção ocorra corresponde ao monitoramento (não é o mesmo monitoramento do QoSP), pois é o mesmo que fornece o *feedback* para saber se a QoS está sendo mantida. O módulo de monitoramento do QoSP deve se comunicar com o mecanismo de controle/gerenciamento das arquiteturas que estão sendo utilizadas no ambiente para conseguir obter a QoS que está sendo provida a um canal.

Entre as diversas arquiteturas propostas na literatura, duas foram padronizadas pela IETF: os Serviços Integrados (*Integrated Services - IntServ*) e os Serviços Diferenciados (*Differentiated Services - Diffserv*). A seguir será apresentada de forma detalhada a arquitetura *Diffserv*, visto que a mesma foi utilizada na implementação do QoSP.

2.1.1 Diffserv

Os Serviços Diferenciados (BLAKE et al., 1998) baseiam-se na idéia de classes de comportamento, sendo que cada classe de comportamento define uma forma diferente de encaminhamento dos pacotes. Os pacotes pertencentes a diversos fluxos são agregados em classes e os pacotes da mesma classe são encaminhados na rede utilizando a mesma classe de comportamento. As reservas de recursos assim como o tratamento dado pelos roteadores (definido pelas classes de comportamento) aos pacotes é feito por classe, tornando o sistema escalável, visto que para um número grande de fluxos são configuradas poucas classes e, conseqüentemente, poucos estados são armazenados em cada roteador. Outra vantagem de se reservar recursos por classe é que raramente os recursos estarão ociosos, visto que frequentemente pelo menos um canal estará transmitindo algum dado. A maior parte do tráfego da Internet se comporta desta maneira, ou seja, nem todos os canais estarão transmitindo ao mesmo tempo.

As classes de comportamento são chamadas de PHB (*Per Hop Behavior*). As PHBs definem a forma como cada roteador irá encaminhar os pacotes para o outro roteador. Desta maneira, cada roteador deve possuir pré-configurado um conjunto de PHBs. Esta forma de encaminhamento se dá em termos dos recursos a serem utilizados e das propriedades do serviço que devem ser consideradas (como perdas, atraso, *jitter*) (GORENDER, 2005). Como podemos observar, a idéia do *Diffserv* é oferecer níveis diferentes de serviço para os pacotes, ou seja, ele se baseia em um esquema de prioridade relativa (XIAO; NI, 1999). O conjunto de fluxo de dados tratados da mesma forma (assinalados à mesma PHB) recebem o nome de Agregado de Comportamento (*Behavior Aggregate*).

Cada roteador identifica a classe de comportamento que será aplicada ao pacote através do *DS Field*. Este corresponde a um campo do cabeçalho IP, e inclui seis bits que são utilizados pelo *Diffserv* como um *Codepoint* (*DSCP*). Cada valor do *DSCP* é mapeado para uma PHB em um roteador.

Para que um determinado cliente possa receber Serviços Diferenciados do provedor de serviços (ISP), é necessário que seja estabelecido um Acordo de Nível de Serviço (*Service Level Agreement - SLA*) entre o cliente e o ISP. Este acordo informa o nível de serviço assim como identifica as características do fluxo de dados que deverá receber o Serviço Diferenciado. O próprio cliente ou o ISP pode marcar o DSCP para configurar o serviço que será recebido pelo cliente.

Um domínio *Diffserv* (Figura 2.1) é formado por roteadores (algumas literaturas chamam de nós) de dois tipos: roteadores de borda (também conhecidos como nós de fronteira) e de núcleo (também conhecidos como nós interiores) (BLAKE et al., 1998). Os roteadores de borda servem para interconectar domínios enquanto que os roteadores de núcleo apenas se conectam a outros roteadores do domínio. Os nós de fronteira implementam as funções de classificação, marcação, policiamento e condicionamento de tráfego, enquanto que os nós interiores apenas classificam, sendo que esta classificação é mais simples visto que ela é baseada apenas no DSCP, e encaminham pacotes. Esta estruturação da rede *Diffserv* contribui para sua escalabilidade.

Existem quatro PHBs padronizadas pelo IETF mas, para que o QoSP funcione (visto que os canais gerenciados pelo QoSP necessitam de dois tipos de serviços), apenas duas

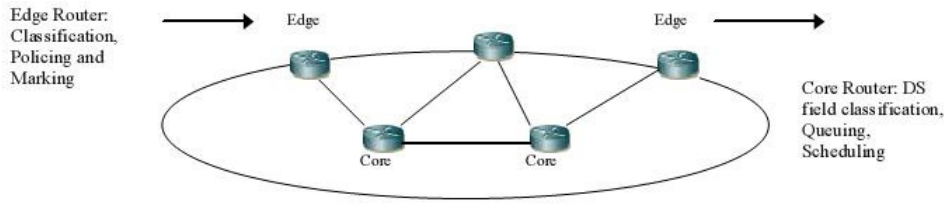

Figura 2.1 Rede Diffserv (KLEITHY; FRISBY; ROGHLú, 2003)

PHBs são necessárias e, consequentemente, serão abordadas: Padrão e o Serviço Expresso. Estes serviços caracterizam os níveis extremos de garantias, visto que enquanto o primeiro não possui garantia nenhuma, o segundo possui garantias suficientes para que o pacote sofra o menor atraso possível nos roteadores.

2.1.1.1 Padrão A PHB Padrão especifica que os pacotes deverão receber o serviço melhor esforço (*best-effort*). Neste caso, os pacotes que possuem tal serviço não têm prioridade e não possuem garantias de recursos. Este serviço será utilizado pelos canais de comunicação *untimely*, que serão descritos no capítulo 3.

2.1.1.2 Serviço Expresso O Serviço Expresso (*Expedited Forwarding* - EF), sendo também chamado em algumas bibliografias por serviço *Premium*, basicamente visa garantir baixa latência e *jitter*, além de uma largura de banda garantida. De forma intuitiva, para que o roteador forneça o Serviço Expresso basta que a taxa de saída dos pacotes seja maior ou igual a uma taxa de entrada previamente configurada (BLAKE et al., 1998). Com isso, caso os fluxos de dados não excedam a taxa de transmissão previamente configurada, praticamente estes pacotes não permanecerão em fila. Este serviço será utilizado pelos canais de comunicação *timely*, descritos no capítulo 3.

2.2 SISTEMAS DE MONITORAMENTO

A monitoração de QoS é um mecanismo utilizado por outros componentes para obter um *feedback* sobre o estado atual da QoS, para que medidas possam ser tomadas baseado neste *feedback*. A reserva dos recursos não é suficiente para se obter QoS, visto que a degradação da mesma é inevitável (JIANG; THAM; KO, 2000). De acordo com (AURRECOECHEA; CAMPBELL; HAUW, 1996), a monitoração da QoS é um componente essencial para o mecanismo de gerenciamento. Ela é responsável por descobrir o nível de QoS que está sendo provido por cada componente do serviço de comunicação para que se possa então verificar uma eventual degradação da qualidade de serviço fornecida. Quando a degradação é detectada, esta deve ser informada à aplicação e, neste caso, a aplicação poderá reagir de maneira adequada à perda da QoS, assumindo a perda no nível do serviço ou tentando renegociar uma nova QoS para o canal de comunicação. De acordo com (AURRECOECHEA; CAMPBELL; HAUW, 1996), o mecanismo de gerenciamento deve tentar resolver a perda da QoS antes de informar à aplicação sobre a degradação. O módulo de monitoração do QoSP deve apenas verificar e informar a QoS que está sendo

provida ao canal, não sendo necessário resolver uma possível perda.

Mesmo o módulo de monitoração do QoSP não sendo um componente de um arquitetura de QoS, ele se comporta como um para o QoSP, visto que ele é responsável por descobrir o nível de QoS que está sendo provido por cada arquitetura de QoS utilizada. Sendo assim, é importante que modelos e princípios de um sistema de monitoramento sejam utilizados.

2.2.1 Princípios

De acordo com (ASGARI et al., 2002), a escalabilidade nas redes IP com QoS está ligada a três fatores: tamanho da topologia da rede, número e granularidade das classes de serviço suportadas na rede e o número de clientes que utilizam os serviços. Um sistema de monitoramento é considerado escalável se o mesmo consegue monitorar de forma eficiente uma grande rede com vários serviços e uma grande quantidade de clientes. Além disso, o sistema de monitoramento deve realizar as seguintes tarefas: coletar, agregar e analisar dados, além de prover um *feedback* sobre a QoS atual. Ainda de acordo com (ASGARI et al., 2002), para que o sistema de monitoramento seja escalável, é preciso que o mesmo siga os princípios descritos abaixo.

1. Definir granularidade do processo de monitoramento

   Os algoritmos de monitoramento devem operar no nível do agregado e não no nível do pacote, visto que coletar dados no nível de pacote é extremamente custoso e não escalável. No Diffserv, estas estatísticas seriam coletadas por PHB.

2. Distribuir o sistema coletor de dados

   Para que o sistema coletor de dados não gere um *overhead* grande na rede, ele deve ser distribuído, preferencialmente um agente coletor por cada roteador e o mais próximo possível do mesmo.

3. Minimizar o *overhead* da transmissão das medições através do processamento dos dados brutos próximo às fontes

   Assim como o sistema de coleta de dados, o processamento e sumarização dos dados também deve ser distribuído para que o sistema seja escalável. Além do mais, o sistema de monitoração pode adotar a política de notificação de eventos quando certos limiares são ultrapassados. A notificação é utilizada para evitar a sobrecarga da rede com interações desnecessárias entre componentes requisitando informações de monitoramento.

4. Utilizar medições por agregado em combinação com medições por canal

   Vários fluxos de dados possuem requisitos diferentes e por isso devem ser monitorados de forma diferente. Mas monitorar os canais de forma diferente torna o sistema complexo. Além disso, vários canais de comunicação passam pelo mesmo caminho na rede, conseqüentemente as medições feitas para um canal podem ser utilizadas para os outros canais, colaborando para a escalabilidade do sistema. Dessa forma

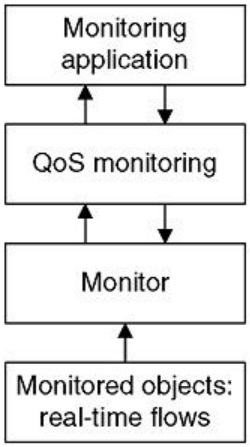

Figura 2.2 Modelo de monitoramento de QoS (JIANG; THAM; KO, 2000)

deve-se optar por uma medição por agregado e, quando isso não for possível devido a requisitos diferentes ou caminhos diferentes pelos quais os canais passam, utiliza-se a medição por canal.

2.2.2 Modelo de monitoramento de QoS

De acordo com (JIANG; THAM; KO, 2000), os mecanismos de monitoramento podem ser divididos em duas categorias, baseado na informação de QoS que pode ser obtida através deles: monitoração de QoS fim-a-fim e a monitoração distribuída de QoS. Na monitoração fim-a-fim, apenas a QoS fim-a-fim de um canal de comunicação é monitorada. Neste caso, a degradação da QoS consegue ser detectada, mas o componente que está causando tal degradação não. Na abordagem distribuída, a QoS que está sendo provida por cada elemento ao longo do canal é monitorada, permitindo saber quais são os elementos que estão causando a degradação. Enquanto que na abordagem fim-a-fim as informações de monitoramento são coletadas nos sistemas fins, ou seja nos *hosts*, na abordagem distribuída estas informações são colhidas dos monitores relevantes, ou seja, dos elementos que estão monitorando os roteadores que fazem parte de um determinado canal. A abordagem distribuída além de contribuir para a escalabilidade permite que medidas de correção sejam tomadas, visto que o problema pode ser isolado. O modelo de monitoramento adotado em (JIANG; THAM; KO, 2000) (Figura 2.2) é semelhante ao modelo tradicional de sistema de monitoramento apresentado em (STALLINGS, 1996), sendo dividido nos seguintes componentes:

Monitoring application: Este componente provê a interface do sistema de monitoramento com o usuário (ou um outro cliente interessado na informação de monitoramento, como o detector de defeitos). Recolhe (agrega) as informações colhidas pelos *monitores* (o componente *Monitor*), analisa estas informações e provê os resultados da análise para o usuário. Estes resultados correspondem à QoS que está sendo provida ao canal de comunicação.

QoS monitoring: Este componente não existe no modelo tradicional. Este compo-

nente provê mecanismos para permitir que *Monitoring application* recupere a informação dos *monitores* relevantes, ou seja, aqueles que monitoram os roteadores que fazem parte do canal de comunicação em questão. A QoS do canal é derivada a partir das informações capturadas pelos *monitores*.

Monitor: É responsável por colher e armazenar as informações provenientes dos roteadores. Estas informações são comunicadas ao *Monitoring application*.

Monitored objects: Estes objetos correspondem aos atributos e atividades que devem ser monitoradas na rede. Estes objetos equivalem a contadores dos fluxos de tempo real monitorados.

Como foi explicado anteriormente, o sistema de monitoramento desempenha um conjunto de funções básicas. Estas funções podem ser mapeadas para o modelo descrito acima da seguinte maneira: a função de coletar os dados fica à cargo de *Monitor*, enquanto que as funções de agregação e análise dos dados, além do *feedback* provido contendo a QoS do canal são de responsabilidade de *Monitoring application*.

2.2.3 Colhendo informações de QoS

Os sistemas de monitoramento precisam se comunicar com os elementos de rede (como os roteadores) para colher informações de QoS. Algumas vezes essas interações são necessárias para que informações mais detalhadas sobre o estado da QoS que está sendo provida pelos elementos sejam capturadas. O protocolo SNMP é utilizado para gerenciar os elementos de rede, sendo descrito a seguir.

2.2.3.1 SNMP O SNMP (*Simple Network Management Protocol*) foi concebido para monitorar os nós (*hosts*, *switches*, roteadores, etc) na Internet (CASE et al., 1990). Ele é formado por três componentes: o agente, o gerente (conhecido também como sistema de gerenciamento) e os dispositivos gerenciados (*hosts*, roteadores, etc). Os agentes são elementos de software instalados nos dispositivos gerenciados cuja função principal é enviar informações relativas aos dispositivos para os gerentes.

Todas as informações gerenciáveis são definidas através de variáveis nos dispositivos gerenciados. Estas variáveis contém valores que são lidos (quando o gerente solicita ao agente uma operação de leitura) ou atualizados (quando o gerente solicita ao agente uma operação de escrita). Essas informações gerenciáveis podem ser: número IP, quantidade de pacotes descartados, tamanho de filas, largura de banda reservada para classes, dentre inúmeras outras. Estas variáveis são estruturadas hierarquicamente (como numa árvore) na MIB.

Uma *Management Information Base* (MIB) contém a definição dos objetos gerenciáveis. Os objetos gerenciáveis são conhecidos como objetos MIB, e são formados por várias instâncias de objeto. Por exemplo, um número de interface em um roteador é um objeto MIB e, sabendo que um roteador é formado por várias interfaces, o número de uma interface específica seria uma instância de objeto. Cada instância de objeto é identificada por um OID (*object identifier*). Cada instância de objeto corresponde à uma variável citada anteriormente.

Capítulo

3

QOS PROVIDER

Como foi descrito na introdução, para que o modelo HA funcione corretamente, ele necessita obter informações sobre a QoS que está sendo provida aos canais de comunicação. Essas informações são obtidas através da infraestrutura de comunicação e gerenciamento de recursos para sistemas distribuídos com QoS denominada *QoS Provider* (QoSP). O objetivo desta infraestrutura é fornecer e gerenciar canais de comunicação providos com QoS (GORENDER, 2005). Para isso, o QoSP é definido através de uma interface padronizada que fornece um conjunto fixo de serviços, descritos nas próximas seções. Através desta padronização, o QoSP visa tornar o modelo citado anteriormente portável para diversas arquiteturas de QoS. Ele pode ser visto como um invólucro para um conjunto de arquiteturas de QoS, sendo assim deve se comunicar com as arquiteturas existentes no ambiente para obter as informações relativas a QoS de um canal de comunicação.

Considera-se que um sistema distribuído é composto por um conjunto de processos, os quais executam em um ou mais *hosts*. Este conjunto é representado por $\Pi = \{p_1, p_2, ..., p_n\}$, sendo p_x um processo qualquer deste conjunto e n o número de processos pertencentes ao sistema. Um canal de comunicação estabelecido entre os processos p_x e p_y é identificado por $c_{x/y}$ e Γ é o conjunto de todos os canais de comunicação estabelecidos.

3.1 AMBIENTE DE EXECUÇÃO

Os canais de comunicação aqui utilizados são definidos como confiáveis e são providos com diferentes níveis de QoS. Canais confiáveis são aqueles que não perdem, não duplicam e nem alteram as mensagens. Os canais de comunicação são estabelecidos entre dois processos localizados em *hosts* diferentes, definidos através de uma rota estabelecida em uma rede de computadores, sendo que esta rota é formada por um ou mais roteadores.

As arquiteturas de QoS padronizaram várias classes de serviço (no capítulo 2 foram descritas duas classes de serviço para a arquitetura Diffserv). Baseado nas diversas classes de serviço padronizadas pelas diversas arquiteturas de QoS, (GORENDER, 2005) definiu duas classes de serviço: Serviços Isócronos e Serviços Não Isócronos. Os Serviços Isócronos são serviços de comunicação que possuem limites temporais determinados para

o processamento e transferência das mensagens. Os serviços Não Isócronos não fornecem nenhum limite temporal tanto para o processamento como para a tranferência das mensagens. Os canais de comunicação que utilizam os Serviços Isócronos são denominados canais *timely* enquanto que os canais que utilizam os Serviços Não Isócronos são denominados canais *untimely*.

3.2 ARQUITETURA DO QOS PROVIDER

O QoS Provider executa distribuído, sendo que existe um módulo do QoSP para cada *host* do sistema. Os módulos do QoSP necessitam trocar mensagens entre si e com os roteadores para que as funções definidas na sua interface possam funcionar corretamente. Quando um processo p_x necessita criar canais de comunicação com QoS e acessar informações de QoS dos mesmos, ele faz isso através do módulo $QoSP_x$, sendo $QoSP_x$ o módulo do QoSP localizado no mesmo *host* de p_x.

Para que o QoSP possa funcionar corretamente, os canais de comunicação utilizados pelos módulos do QoSP também necessitam utilizar os serviços das arquiteturas de QoS, discutidos no tópico anterior. Isto se faz necessário visto que os módulos do QoSP precisam se comunicar para realizar suas tarefas e, para que as informações providas pelos mesmos sejam confiáveis, eles também necessitam de reservas de recursos. Para isso, o QoSP também utilizará os serviços da arquitetura *Diffserv* e um sistema operacional de tempo real instalado nos *hosts* (o sistema operacional de tempo real utilizado na implementação do QoSP e pelos processos clientes será apresentado no capítulo 4).

3.3 INTERFACE DO QOS PROVIDER

De acordo com (GORENDER, 2005), a interface do QoSP pode ser definida através das seguintes funções:

- $CreateChannel(p_x, p_y) : \Pi^2 \rightarrow \Gamma$

  Cria um canal de comunicação entre os processos p_x e p_y, retornando o novo canal, identificado por $c_{x/y}$, pertencente ao conjunto Γ. Todo canal deve ser criado como *untimely*. As informações referentes ao canal, como a sua QoS, devem ser armazenadas pelo QoSP.

- $DefineQoS(p_x, p_y, qos) : \Pi^2 \times \{timely, untimely\} \rightarrow \{timely, untimely\}$

  Altera a QoS provida ao canal de comunicação $c_{x/y}$ entre *timely* e *untimely*. Caso a solicitação seja de *untimely* para *timely*, um processo de admissão deverá ser executado.

- $Delay(p_x, p_y) : \Pi^2 \rightarrow N^+$

  Calcular um limite máximo, caso o canal seja *timely*, ou um limite probabilístico, caso o canal seja *untimely*, para o tempo de ida e volta (RTT maior ou igual ao *Round trip time*) de transferência de uma mensagem entre os processos p_x e p_y.

- $QoS(p_x, p_y) : \Pi^2 \rightarrow \{timely, untimely\}$

Verificar a QoS que está sendo provida ao canal de comunicação $c_{x/y}$, retornando a classe de serviço (*timely* ou *untimely*) que está sendo provida ao mesmo. Quando a QoS de um canal for alterada de *timely* para *untimely*, os processos ligados ao canal devem ser informados de tal degradação.

- $VerifyChannel(c_{x/y}) : \Gamma \rightarrow \{timely, untimely\}$

  Assim como a função *QoS*, esta função tem como objetivo verificar a QoS que está sendo provida a um canal de comunicação ($c_{x/y}$). Uma das diferenças com relação à função *QoS* é que ela é chamada por um processo p_i, sendo $p_i \neq p_x$ e $p_i \neq p_y$ (mais detalhes sobre esta função será explicado na próxima seção). A outra diferença é que esta função além de verificar a QoS que está sendo provida ao canal, através da chamada da função *QoS* por exemplo, verifica a existência de tráfego no canal $c_{x/y}$. A não existência de tráfego durante um certo intervalo de tempo no canal (sendo que este intervalo de tempo é estabelecido pelo modelo HA) faz com que a QoS do canal seja alterada para *untimely*.

Além das funções citadas acima, cada módulo do QoSP deve monitorar de forma automática todos os canais *timely* gerenciados pelo mesmo, com o intuito de verificar se estes canais continuam sendo providos com serviço Isócrono. Esta monitoração deve ocorrer periodicamente e, caso seja detectado que a QoS de um canal alterou de *timely* para *untimely*, os processos ligados ao canal devem ser notificados de tal degradação.

3.4 O MECANISMO DE MONITORAMENTO DO QOSP

Como já foi dito na introdução, apesar da interface do QoSP ser definida através de cinco funções (descritas na seção anterior), ela pode ser resumida em dois grandes serviços: negociação e monitoração. Enquanto que o serviço de negociação pode ser considerado um mecanismo estático de QoS, o serviço de monitoração pode ser considerado um mecanismo dinâmico (AURRECOECHEA; CAMPBELL; HAUW, 1996). Enquanto o serviço de negociação visa basicamente a reserva dos recursos, o serviço de monitoração visa garantir que os processos aplicativos tenham um *feedback*, o quanto mais cedo possível, sobre o serviço que está sendo provido aos seus canais de comunicação. Este *feedback* pode ser obtido através de uma solicitação explícita por parte dos processos aplicativos, ou através de uma notificação provida pelo mecanismo de monitoramento aos processos aplicativos, quando há uma alteração de QoS dos seus canais.

Baseado no que foi descrito acima, o mecanismo de monitoramento do QoSP, também conhecido como *QoS Provider Monitoring* (QoSPM), engloba as seguintes funções da interface do QoSP: *QoS* e *VerifyChannel*. Ambas procuram fornecer um *feedback* sobre a QoS atual do canal para que medidas possam ser tomadas. Além destas funções, o monitoramento automático também faz parte do QoSPM, sendo ele equivalente à chamada periódica da função *QoS*.

A função *QoS* executa a verificação da QoS de um canal $c_{x/y}$, sendo p_x o solicitante da verificação. Esta é efetuada através do envio de mensagens de verificação a todos os roteadores pertencentes a $c_{x/y}$ e ao módulo $QoSP_y$, além do estabelecimento de *timeouts* para a recepção das respostas. O módulo do QoSP (mas especificadamente o $QoSP_x$) fica

aguardando as respostas de todas as mensagens enviadas. Se todas as mensagens chegam antes de *timeout* (calculado através da função *Delay*) e indicam que o serviço Isócrono está sendo provido, assume-se que a QoS do canal é *timely*. Em qualquer outra situação, assume-se que a QoS do canal é *untimely*. Caso a função *QoS* mostre que a QoS do canal foi alterada de *timely* para *untimely*, os processos ligados ao canal devem ser informados de tal alteração.

A função *VerifyChannel* também verifica a QoS de um canal $c_{x/y}$, sendo p_i o processo solicitante da verificação. Como $p_i \neq p_x$ e $p_i \neq p_y$, caso p_i não esteja no mesmo *host* de p_x nem de p_y, o módulo $QoSP_i$ deverá solicitar uma verificação remota ou ao módulo $QoSP_x$ ou ao módulo $QoSP_y$. Independente do módulo que execute a verificação, ela é efetuada da seguinte maneira: executa-se a função *QoS* e, caso o retorno da função seja *untimely*, o retorno da função *VerifyChannel* também será *untimely*. Caso contrário (o retorno de *QoS* seja *timely*), será verificado se os processos p_x e p_y trocam mensagens nos próximos Δt (sendo Δt estabelecido pelo modelo HA). Caso ocorra troca de mensagens neste intervalo de tempo, o retorno da função *VerifyChannel* será *timely*, caso contrário será *untimely*.

Assim como os processos aplicativos interagem com o QoSPM, o módulo de negociação do QoSP também precisa interagir com o QoSPM. Quando a QoS de uma canal é alterada (através da função *DefineQoS*), esta informação precisa ser passada ao QoSPM.

Capítulo 4

SISTEMA OPERACIONAL DE TEMPO REAL

De acordo com (VERíSSIMO; RODRIGUES, 2001), sistemas de tempo real podem ser definidos como sendo aqueles cuja progressão é especificada em termos dos requisitos de *timeliness* ditados pelo ambiente. *timeliness* é uma propriedade que especifica que um determinado predicado ρ será verdadeiro em dado instante de tempo. A partir desta definição, podemos observar que sistemas de tempo real estão relacionados com previsibilidade. Esta previsibilidade é necessária para que tanto os canais *timely* (descritos no capítulo 3) possam ser providos como também para que estes possam ser gerenciados da maneira correta pelo QoSP.

Nas seções seguintes serão descritas as abordagens utilizadas para se desenvolver os sistemas operacionais de tempo real (*Real-time Operating Systems* - RTOS), sendo que nas duas últimas seções serão descritos os dois *frameworks* de tempo real utilizados pelo QoSP.

4.1 ABORDAGENS DE RTOS

De acordo com (YODAIKEN; BARABANOV, 1997), existem três abordagens principais para construir um sistema operacional de tempo real: construir um sistema operacional específico para tempo real, adicionar suporte a tempo real em um sistema operacional de propósito geral (SOPG) e, por último, utilizar um sistema operacional de tempo real juntamente com um de propósito geral em um mesmo computador. A primeira abordagem é pouco utilizada visto que a maioria dos drivers são desenvolvidos para sistemas operacionais de uso geral, limitando a utilização destes sistemas operacionais específicos. As duas outras abordagens serão discutidas a seguir. Apesar destas abordagens não serem específicas para um determinado projeto, nos projetos pesquisados, estas são desenvolvidas utilizando como base o Linux. Isto se explica devido ao fato do Linux ser um sistema operacional livre, com o código fonte disponível. Sendo assim, em algumas situações a expressão *sistema operacional de propósito geral* e a palavra *Linux* serão utilizados de forma intercambiável.

4.1.1 SOPG com suporte de tempo real

Esta abordagem, também conhecida como *Preemption Improvement*, visa diminuir o tamanho das seções de código não preemptável, além de transformar o *kernel* em versão totalmente preemptável (preemptável se refere à capacidade do sistema operacional de liberar um recurso em posse de um processo). Com isso, tratadores de interrupção e seções críticas passam a ser preemptáveis. O menor tempo de latência de escalonamento (sendo que esta latência se refere ao tempo decorrido entre a ocorrência de um evento externo e o início de execução da tarefa) está relacionado diretamente com o maior trecho de código não preemptável, por isso, limitar o tamanho deste trecho é importante para que restrições de tempo (*soft* ou *hard*) possam ser alcançadas. O *patch* (arquivo que contém um código a ser adicionado em um outro código fonte de programa visando corrigir um *bug* ou adicionar alguma(s) funcionalidade(s)) de preempção de tempo real (PREEMPT_RT) (MCKENNEY, 2005) para o *kernel* do Linux desenvolvido por Ingo Molnar utiliza esta abordagem.

4.1.2 SOPG e RTOS compartilhando o mesmo computador

Nesta abordagem, também conhecida como *Interrupt abstraction*, um *kernel* de tempo real executa independente do *kernel* do linux. A idéia é que os dois ambientes (Linux e RTOS) executem lado a lado, utilizando um *micro-kernel* que gerencia as interrupções geradas pelo hardware (GERUM, 2004). O *kernel* do linux passa a ter prioridade mais baixa, comparada com os processos gerenciados pelo RTOS. A expressão *Interrupt abstraction* se deve ao fato de que o *micro-kernel* cria uma abstração de interrupção (também conhecida como cano de interrupção), onde as interrupções chegam primeiro para o ambiente de tempo real e, caso não pertençam a este ambiente, passam para o Linux. Os principais projetos que utilizam esta abordagem são: Real-Time Linux (YODAIKEN; BARABANOV, 1997), RTAI (RTAI,) e Xenomai (GERUM, 2004).

4.2 XENOMAI

O Xenomai é um *framework* de desenvolvimento de tempo real, que executa de forma cooperativa com o *kernel* do Linux, com o intuito de prover aos programas aplicativos suporte de tempo real rígido (XENOMAI,). De acordo com (GERUM, 2004), o objetivo inicial do Xenomai é permitir aos desenvolvedores portar seus programas de tempo real, que executam em um RTOS específico, para o ambiente GNU/Linux sem a necessidade de reescrever suas aplicações completamente.

A portabilidade dos programas aplicativos, citada anteriormente, é possível graças ao fato do Xenomai ser agnóstico com relação à API que está sendo utilizada, visto que ele permite que os processos de tempo real utilizem diversas APIs para se comunicar com o seu núcleo. O Xenomai é formado por um núcleo que exporta um conjunto genérico de serviços. Estes serviços são agrupados em uma interface de alto nível, permitindo a implementação de vários módulos que emulam diversos RTOS, tais como VxWorks, pSOS, uITRON, VRTX, entre outros (Figura 4.1). Este conjunto de serviços também foi utilizado para implementar uma API nativa e uma API em conformidade com o padrão

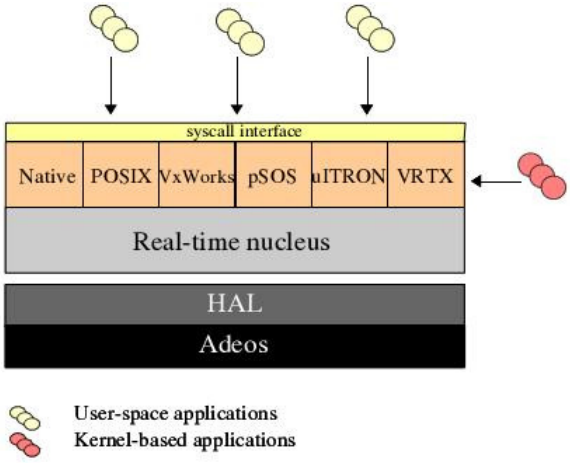

Figura 4.1 Skins para o Xenomai (XENOMAI, 2006)

POSIX.

O Xenomai utiliza como *micro-kernel* o ADEOS (YAGHMOUR, 2001), uma camada de abstração do hardware que permite que vários sistemas operacionais compartilhem os recursos de hardware (Figura 4.1). O ADEOS trabalha com o conceito de domínios, que são as entidades que coexistem na máquina, onde cada domínio não tem o conhecimento do outro mas todos se comunicam com o ADEOS (YAGHMOUR, 2001).

As aplicações de tempo real que são desenvolvidas desde o início tendo como base o Xenomai, geralmente utilizam a API nativa (XENOMAI, 2006) do Xenomai. Esta oferece vários serviços disponibilizados comumente nos RTOS (VERíSSIMO; RODRIGUES, 2001) (KALINSKY, 2007). Os serviços são divididos em seis grandes categorias, sendo que as principais são: gerenciamento de tarefas, serviços de tempo, suporte à sincronização, além de comunicação e transferência de mensagens. Estes serviços podem ser chamados tanto no contexto do espaço do kernel como no contexto do espaço de usuário, permitindo que as aplicações possam executar nos dois contextos.

O Xenomai permite que as tarefas de tempo real executem tanto no domínio do Xenomai (modo primário) como no domínio do Linux (modo secundário), de forma totalmente transparente para os desenvolvedores. Quando uma tarefa executando no domínio do Xenomai faz uma chamada de sistema ao Linux, esta migra para o domínio do Linux, passando a ser gerenciado por este. Quando a execução da chamada de sistema é finalizada, a tarefa volta para o domínio do Xenomai. Para que a previsibilidade possa continuar a ser provida mesmo quando uma tarefa está sob gerência do kernel do Linux, o Xenomai disponibiliza o escudo de interrupção (*Interrupt shield*) (XENOMAI, 2006).

Para que um sistema distribuído possa ter suas restrições de tempo atendidas, além de um kernel de tempo real, é necessário que o sistema de comunicação na rede também forneça garantias temporais. O projeto RTnet, descrito a seguir, visa atender tais requisitos.

4.3 RTNET

O RTnet é um *framework* para comunicação de tempo real rígido sobre Ethernet e outros meios de comunicação (KISZKA et al., 2005). Enquanto o Xenomai (apresentado anteriormente) se preocupa basicamente com o escalonamento das tarefas de tempo real (está relacionado com o processamento das mensagens), o RTnet foca a transferência das mensagens sobre uma rede Ethernet com restrições rígidas de tempo. Neste caso, podemos observar que o Xenomai e o RTnet se complementam na função de fornecer uma solução para a comunicação em rede que atenda às restrições rígidas de tempo.

Basicamente o RTnet consiste em uma pilha de protocolos de rede capaz de prover comunicação de tempo real sobre Ethernet. Para garantir as restrições rígidas de tempo, o RTnet implementa os protocolos UDP/IP, ICMP e ARP retirando todas as possíveis causas de indeterminismo (KISZKA et al., 2005).

Além da implementação dos protocolos de forma determinística, o RTnet provê um controle de acesso ao meio determinístico através de sua camada de acesso ao meio denominada RTmac. Com o RTmac, as aplicações de tempo real conseguem obter QoS a um custo baixo, visto que não dependem que um *hardware* implemente a solução de QoS (KISZKA et al., 2005). Por padrão, a disciplina de acesso ao meio utilizada pelo RTnet para se comunicar na rede Ethernet é o TDMA (*Time Division Multiple Access*). Esta disciplina segue a abordagem mestre-escravo, onde o nó mestre no segmento Ethernet é responsável por sincronizar os relógios dos nós escravos e por definir o momento em que estes podem enviar seus pacotes.

Capítulo 5

ESPECIFICAÇÃO DO QOSPM

Neste capítulo o QoSPM será especificado, sendo que ele está dividido em quatro seções: na seção 5.1 será descrita a arquitetura do QoSPM; na 5.2 a modelagem do QoSPM através da UML será abordada; na 5.3 será descrito o protocolo utilizado pelos componentes do QoSPM; e para finalizar, a seção 5.4 abordará os principais algoritmos utilizados pelo QoSPM.

5.1 ARQUITETURA DO QOSPM

A arquitetura do QoSP foi descrita no caítulo 3. Agora nós iremos focar a arquitetura do QoSP *Monitoring*. Como foi descrito anteriormente, uma rede com o QoS *Provider* é formada por módulos do QoSP e por roteadores. Além destes elementos, um novo elemento foi adicionado para compor a arquitetura do QoS *Provider*, mas especificadamente do QoSPM, sendo denominado QoSP *Agent* (QoSPA). Para cada roteador existe um módulo do QoSPA, sendo que este tem a função de monitorar o roteador que está vinculado ao mesmo (Figura 5.1).

5.1.1 O QoSP

Os módulos do QoSP executam nos *hosts*, sendo que estes *hosts* são aqueles onde se localizam os componentes que caracterizam o modelo HA, assim como os processos distribuídos que executam sobre o modelo HA. Além dos módulos do QoSP trocarem mensagens entre si (como mensagens de requisição remota de verificação de canal), um determinado QoSP comunica-se com vários módulos do QoSPA para colher informações sobre a QoS que está sendo provida para um determinado canal (mas especificadamente para uma determinada classe de serviço da arquitetura Diffserv), cujos roteadores estão sendo monitorados pelos módulos do QoSPA com os quais a comunicação foi estabelecida. Depois de agregada as informações, a QoS do canal pode ser então descoberta. O QoSP também se comunica com o QoSPA para informar sobre seu interesse em receber informações relativas à degradação da QoS no roteador monitorado pelo QoSPA, com o intuito de antecipar

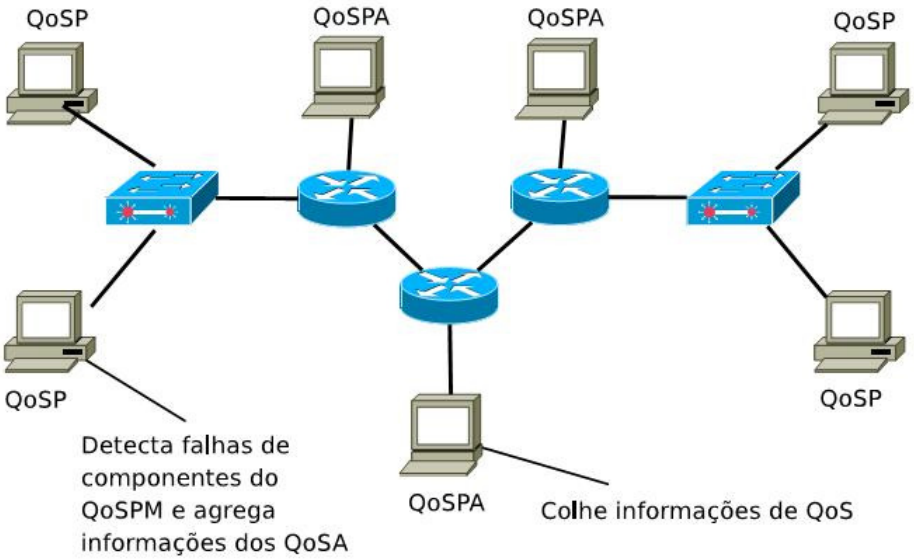

Figura 5.1 Arquitetura do QoSPM

o *feedback* provido pelo mecanismo de monitoramento aos processos aplicativos. Sendo assim, a política de notificação de eventos é adotada (ver 2).

A arquitetura utilizada pelo QoSP segue a abordagem distribuída adotada pelo modelo de monitoramento descrito no capítulo 2. Os componentes *Monitoring Application* e *QoS Monitoring* do modelo de monitoramento fazem parte do QoSP. Além das funções descritas anteriormente, o QoSP é responsável pela detecção de falhas dos elementos que são importantes para o funcionamento do QoSPM (roteadores, módulos do QoSP e módulos do QoSPA).

5.1.2 O QoSPA

Cada módulo do QoSPA está vinculado a um roteador específico e sua função é colher informações sobre QoS no roteador vinculado ao mesmo. O QoSPA executa em um *host* conectado diretamente ao roteador, seguindo um dos princípios de sistemas de monitoramento. Caso o QoSPA não possa executar em *host* específico, o mesmo poderá executar em um *host* onde se localiza um módulo do QoSP. O QoSPA colhe periodicamente informações de QoS, pré-processa e sumariza estas informações que serão utilizadas pelos módulos do QoSP. Neste contexto, o QoSPA funciona como um agente para o QoSP. O componente *Monitor* do modelo de monitoramento faz parte do QoSPA. Seguindo os princípios de sistemas de monitoramento, o QoSPA executa distribuído (um para cada roteador), além das informações colhidas pelos mesmos serem no nível do agregado (no nível das classes do *Diffserv*).

O protocolo utilizado pelo QoSPA para colher informações de QoS é o SNMP. Uma das vantagens de se utilizar o SNMP é que dependendo da granularidade proporcionada pela MIB utilizada para obter informações de QoS, quanto mais específicas forem as variáveis definidas na MIB, menos mensagens serão trocadas entre o QoSPA e o roteador

para se obter uma informação mais robusta. O QoSPA utiliza o SNMP para verificar se a QoS que havia sido previamente negociada para um canal continua sendo provida, verificando se a classe Serviço Expresso configurada no roteador continua provendo seus serviços da maneira esperada. Caso seja percebido que ocorreu degradação, todos os módulos do QoSP interessados em receber tal informação serão notificados.

5.2 MODELAGEM DO QOSPM

A especificação do QoSPM foi feita através da modelagem do sistema utilizando a abordagem orientada à objetos, sendo que tal modelagem foi feita utilizando-se a UML (BOOCK; RUMBAUGH; JACOBSON, 1999). Com a modelagem, além de ser possível especificar a estrutura e comportamento do sistema, as decisões tomadas são documentadas (BOOCK; RUMBAUGH; JACOBSON, 1999).

Baseado na arquitetura descrita na seção anterior e na funcionalidade provida pelo mecanismo de monitoramento, a estrutura do QoSPM foi especificada através de diagramas de classe. Foram desenvolvidos diagramas de classe para modelar tanto o QoSP como o QoSPA.

5.2.1 QoSP

Por ser o módulo principal, o QoSP envolveu tanto a modelagem do domínio como a modelagem da aplicação. O domínio representa o vocabulário inerente ao mecanismo de monitoramento do QoS *Provider*. As informações do vocubulário que necessitam serem armazenadas são relativas aos canais de comunicação. O modelo de classes do domínio (Figura 5.2) é formado pelas seguintes entidades:

- **CommunicationEntity**: Representa qualquer entidade com a qual se possa estabelecer uma comunicação na rede. Essa entidade pode ser tanto um componente de *software* como um componente de *hardware*. Para que essa comunicação seja possível, essa entidade necessita ser localizada através de um endereço IP.
- **Address**: Representa um endereço IP de uma *CommunicationEntity*.
- **QoSPEntity**: Limita o escopo do que uma *CommunicationEntity* pode ser, mas especificadamente, especifica as entidades que compõem o QoSPM e que, possivelmente, necessitarão ser monitoradas.
- **QoSP**: Representa um módulo do QoSP.
- **QoSPA**: Representa um módulo do QoSPA.
- **Router**: Representa um roteador.
- **Channel**: Representa um canal lógico, formado por dois processos e por roteadores.
- **QoS**: Representa a QoS de um *Channel*. No contexto do QoSPM, a QoS de um canal só pode ser *TIMELY* ou *UNTIMELY*.

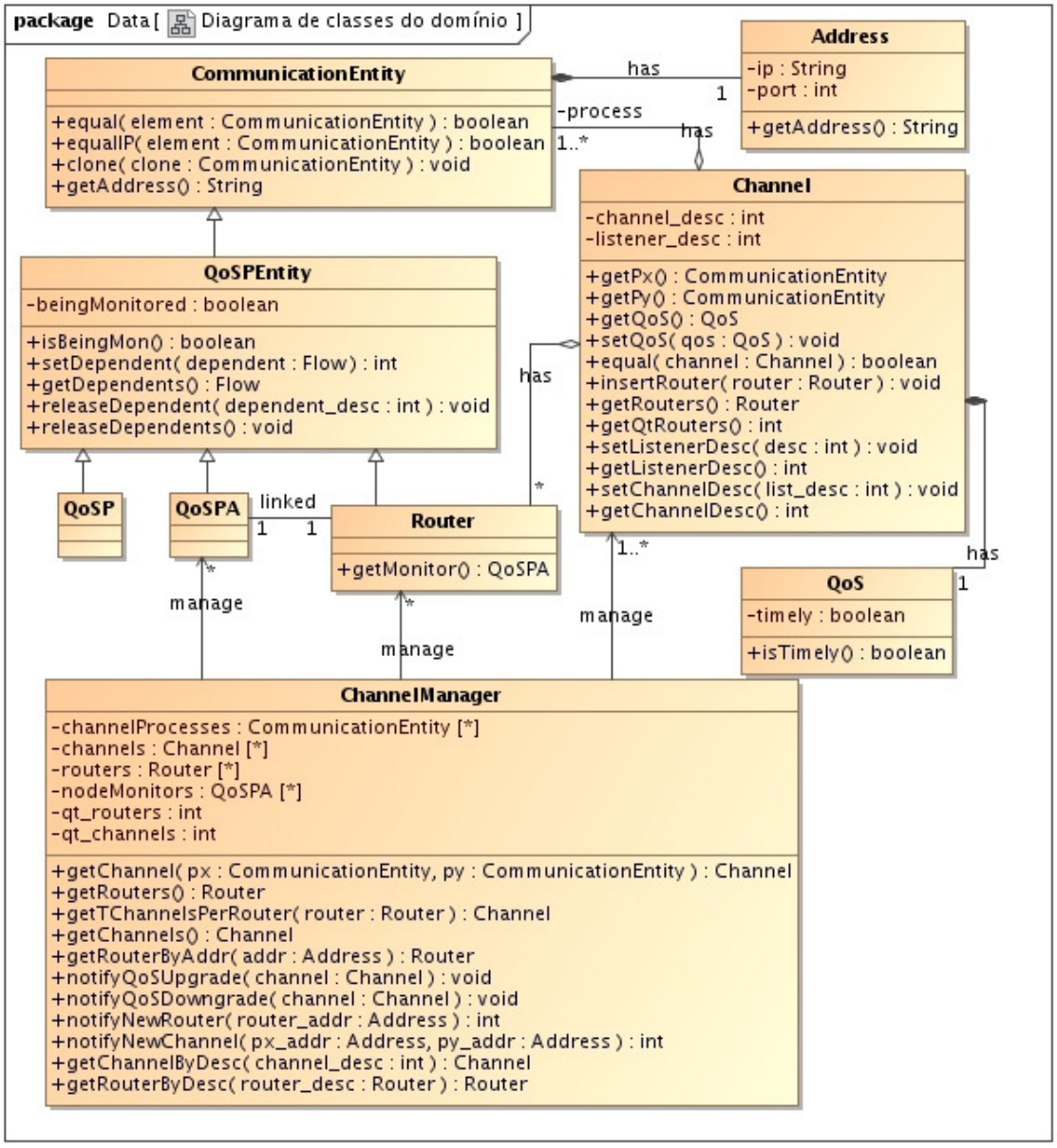

Figura 5.2 Diagrama de classes do Domínio

- **ChannelManager**: Gerencia o principal recurso do QoS *Provider*, os canais, além dos componentes do QoSPM cujas informações necessitam serem recuperadas pelas classes do modelo de aplicação. É importante salientar que *ChannelManager* gerencia apenas os recursos que, de certa forma, estão relacionados aos canais de comunicação cujos processos localizam-se em seu *host*.

O modelo de aplicação define a aplicação (mecanismo de monitoramento) propriamente dita, e não os objetos (definidos no modelo de domínio) sobre os quais a aplicação atua (BLAHA; RUMBAUGH, 2006). A maioria das classes deste modelo são orientadas a computação, diferentemente do modelo de domínio. O modelo de classes da aplicação (Figura 5.3) é formado pelas seguintes entidades:

- **API**: Representa a interface do QoSPM. Como foi descrito no capítulo 3, tanto os processos aplicativos como o módulo de negociação necessitam interagir com o QoSPM. Ambos enxergam o QoSPM através desta classe.

- **Mediator**: Encapsula como os objetos das classes *API*, *MonitoringApplication* e *ChannelVerifier* interagem quando há altereção de QoS de um canal. Esta classe modela o principal participante do padrão de projeto *Mediator* (GAMMA et al., 1995).

- **MonitoringApplication**: É responsável por delegar cada monitoração de QoS para um *QoSMonitor*. Além disso ela também é responsável por verificar quando um determinado módulo do QoSP não necessita mais ser monitorado pelo *FailureDetector*, visto que não existem mais canais passando pelo *host* do mesmo.

- **QoSMonitor**: É responsável por monitorar a QoS de um canal, quando tal monitoramento é requisitado.

- **FailureDetector**: Responsável por monitorar periodicamente objetos *QoSPEntity* à procura de falhas. Caso falhas sejam detectadas, objetos *QoSMonitor* assim como objetos *Verifier* podem usufruir de tal informação, visto que, possivelmente, respostas das mensagens enviadas pelos mesmos não chegarão. É importante salientar que *FailureDetector* é a única classe que necessita estabelecer *timeouts* para a recepção de respostas às suas mensagens enviadas. Tanto *QoSMonitor* como *Verifier* não necessitam estabelecer *timeouts* para a execução de suas funções, visto que eles contam com informações providas por *FailureDetector* sobre falhas em componentes que impediriam suas respostas de chegarem.

- **ChannelVerifier**: É responsável por delegar cada verificação de canal para um *Verifier*. Além disso ela também é responsável por informar *TrafficListener* quando canais de comunicação devem ou não ser escutados.

- **Verifier**: É responsável por realizar a verificação de canal (descrita no capítulo 3), quando tal verificação é requisitada.

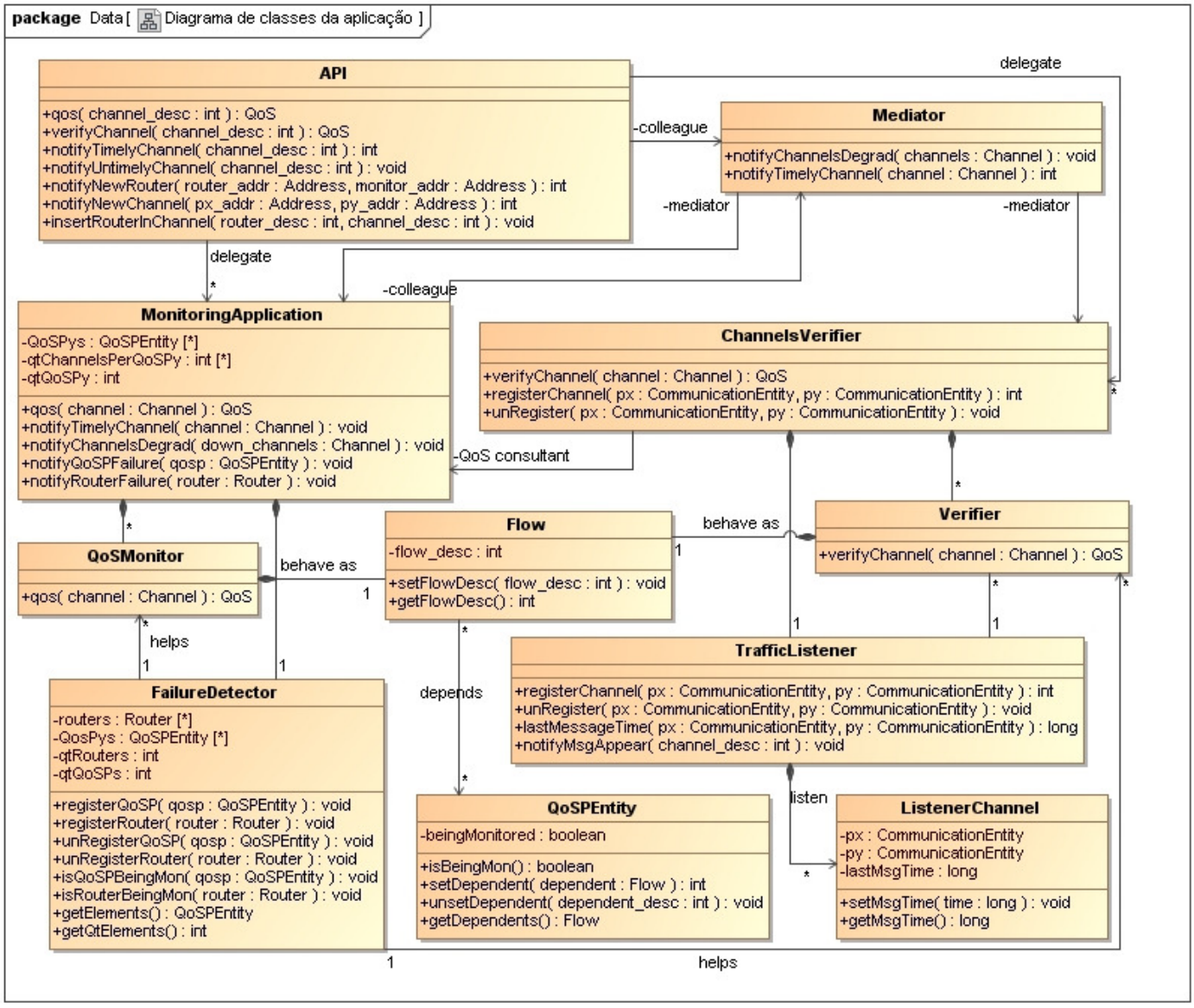

Figura 5.3 Diagrama de classes do QoSP

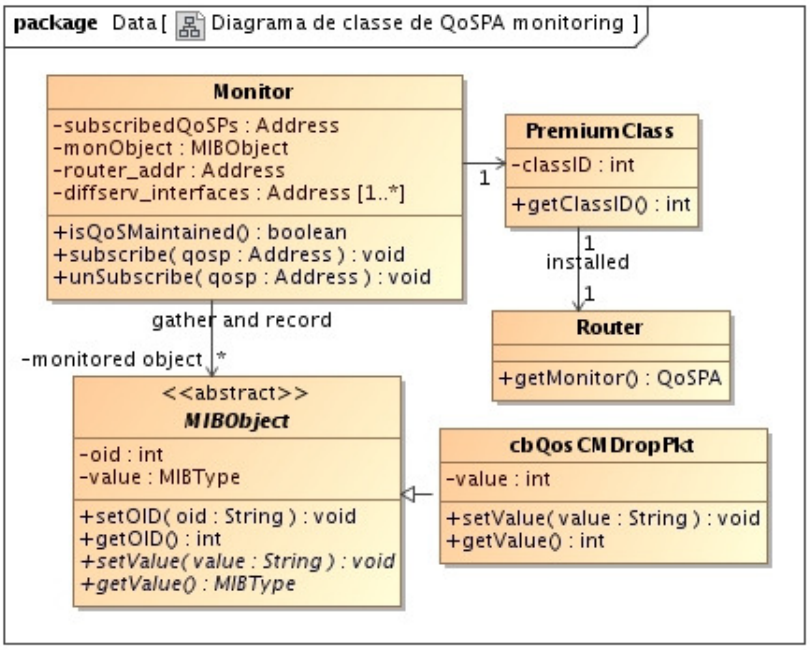

Figura 5.4 Diagrama de classes do QoSPA

- **TrafficListener**: Responsável por escutar os canais de comunicação e armazenar informações sobre o momento em que houve tráfego nestes canais.
- **ListenerChannel**: Representa um canal de comunicação que está sendo escutado por *TrafficListener*.
- **Flow**: Modela o comportamento de um fluxo de execução. Tanto *QoSMonitor* como *Verifier* se comportam como um fluxo de execução, visto que tanto a função de monitoração de QoS como a função de verificação de canal podem ter mais de uma requisição sendo realizada ao mesmo tempo.

5.2.2 QoSPA

O módulo do QoSPA tem uma funcionalidade bem mais restrita se comparado ao módulo do QoSP. O diagrama de classes (Figura 5.4) do mesmo é bem simples, sendo formado pelas seguintes classes:

- **Monitor**: É responsável por colher informações, provenientes do roteador vinculado ao módulo QoSPA do qual faz parte, relativas à classe do serviço Expresso. Essas informações serão utilizadas para verificar se a QoS anteriormente negociada continua sendo provida. Essas informações colhidas correspondem aos *MIBObject*s.
- **MIBObject**: Representa um objeto MIB (explicado na seção que descreve o SNMP no capítulo 2) colhido por *Monitor*. Cada *MIBObject* corresponde a um *Monitored object* do modelo conceitual de monitoramento (descrito no capítulo 2). Essa classe

é abstrata pois o tipo do valor de cada objeto MIB pode variar. O objeto MIB capturado deve implementar os métodos abstratos desta classe.

- **PremiumClass**: Representa a classe de serviço expresso configurada em um roteador.

5.3 PROTOCOLO UTILIZADO PELO QOSPM

Como foi descrito nas seções anteriores, o QoSPM é composto por módulos do QoSP e do QoSPA que trocam mensagens entre si e com os roteadores. Para descrever tais mensagens, foi definido um protocolo de comunicação. A seguir serão descritas as mensagens que fazem parte do protocolo, sendo que para cada mensagem serão descritos o componente que enviou tal mensagem, o componente receptor, em qual contexto assim como a semântica da mensagem enviada, os campos que compõem a mensagem (toda mensagem possui um campo que identifique o tipo da mensagem, sendo assim este não será citado) e, se houver a necessidade, a ordem de precedência com relação a outras mensagens. A palavra *QoSP* será utilizada de forma intercambiável com a expressão *módulo do QoSP* assim como a palavra *QoSPA* será utilizada de forma intercambiável com a expressão *módulo do QoSPA*, ambos os módulos descritos na seção inicial deste capítulo.

- **Monitoring Request**: Esta mensagem é enviada por um QoSP para um QoSPA quando uma requisição de monitoramento (função *QoS* descrita no capítulo 3) de QoS é executada. Requisita ao QoSPA que verifique junto ao roteador vinculado ao mesmo se a classe Serviço Expresso configurada neste roteador continua provendo seus serviços da maneira esperada. Tal mensagem possui um campo que identifica uma requisição de monitoramento específica, visto que várias requisições de monitoramento podem ser feitas ao mesmo tempo. Além disso, ela possui um campo que identifica unicamente uma mensagem deste tipo, visando descartar mensagens antigas.

- **Monitoring Reply**: Esta mensagem é enviada por um QoSPA para um QoSP em resposta à mensagem *Monitoring Request* previamente enviada. Depois que QoSPA verifica junto ao roteador vinculado ao mesmo se a classe Serviço Expresso configurada neste roteador continua provendo seus serviços da maneira esperada, ele manda a resposta ao QoSP. Tal mensagem possui um campo que informa se a QoS está mantida, além de dois outros campos cujos valores e tipos são os mesmos da mensagem *Monitoring Request* recebida.

- **Monitoring Reply Ack.**: Esta mensagem é enviada por um QoSP para um QoSPA em resposta à mensagem *Monitoring Reply* previamente recebida, sendo que na mensagem *Monitoring Reply* consta que houve degradação da QoS. Depois que o QoSP recebeu do QoSPA a resposta do monitoramento e descobriu que houve degradação, ele deve enviar uma mensagem de reconhecimento. Isto é importante pois como ocorreu degradação, a resposta do monitoramento pode não ter sido entregue ao QoSP. Tal mensagem possui dois campos cujos valores e tipos são os mesmos dos dois últimos campos de *Monitoring Reply*.

- **Verification Request**: Esta mensagem é enviada por um QoSP para outro QoSP quando uma requisição de verificação de canal (função *Verify Channel* descrita no capítulo 3) é executada, sendo que o canal em questão não é gerenciado pelo QoSP local (uma requisição remota deve ser executada). Tal mensagem requisita ao QoSP remoto que faça uma verificação local do canal em questão. Tal mensagem possui um campo que identifica os dois processos (através dos endereços IP) que fazem parte do canal a ser verificado. Além disso, esta mensagem possui dois outros campos semelhantes aos campos de *Monitoring Request*.

- **Verification Reply**: Esta mensagem é enviada por um QoSP para outro QoSP em resposta à mensagem *Verification Request* previamente enviada. Depois que o QoSP verifica localmente um canal (através da execução da função *Verify Channel*), ele manda a resposta ao QoSP que havia solicitado a verificação remota. Tal mensagem possui um campo com o resultado da verificação local, além de dois outros campos cujos valores e tipos são os mesmos dos dois últimos campos de *Verification Request*.

- **Subscribe**: Esta mensagem é enviada por um QoSP para um QoSPA quando um canal teve sua QoS alterada para *TIMELY*, sendo que o QoSPA em questão ainda não havia recebido tal mensagem anteriormente. O QoSP envia tal mensagem ao QoSPA para informar que está interessado em receber informações sobre degradação da QoS provida pelo roteador no qual o QoSPA está vinculado. O intuito desta mensagem é de antecipar a informação sobre degradação da QoS, consequentemente antecipar o *feedback* provido pelo QoSPM. Esta mensagem não possui nenhum campo adicional.

- **Unsubscribe**: Esta mensagem é enviada por um QoSP para um QoSPA quando a QoS de todos os canais, que passam pelo roteador vinculado ao QoSPA em questão e que são gerenciados pelo QoSP, foi rebaixada para *UNTIMELY*. O QoSP envia tal mensagem ao QoSPA para informar que não está mais interessado em receber informações sobre degradação da QoS provida pelo roteador no qual o QoSPA está vinculado. Esta mensagem não possui nenhum campo adicional.

- **Notify Degradation**: Esta mensagem é enviada por um QoSPA para um(ns) QoSP(s) quando, depois que informações de QoS são colhidas e pré-processadas, percebe-se que houve degradação da QoS provida pelo roteador. Tal mensagem é enviada para todos os QoSPs que estão interessados em receber informação de degradação de QoS no roteador vinculado ao QoSPA em questão. Esta mensagem não possui nenhum campo adicional.

- **Are you alive**: Esta mensagem é enviada por um QoSP (mas especificadamente pelo detector de falhas do QoSP) para um QoSP/QoSPA/roteador periodicamente. Tal mensagem tem por objetivo verificar se um determinado componente do QoSPM falhou. Esta mensagem equivale a um *Echo Request* do protocolo ICMP. Possui um campo que identifica unicamente uma mensagem deste tipo, além de um campo que identifica o elemento (QoSP ou QoSPA ou roteador) para o qual está sendo enviada

a mensagem. Este campo é necessário para que o detector de falhas possa saber quais os elementos que responderam a esta mensagem dentro do limite temporal.

- **Explicit ping**: Esta mensagem é semelhante à *Are you alive*, mas ela é enviada para um QoSP quando uma requisição de monitoramento de QoS é executada. O QoSP para o qual a mensagem é enviada localiza-se no mesmo *host* de p_y, sendo p_y o processo remoto do canal cuja monitoração foi solicitada. Esta mensagem, em complemento às mensagens *Monitoring Request* enviadas, são necessárias para a execução do monitoramento de QoS. Esta mensagem não possui um campo que identifica o elemento para o qual a mensagem foi enviada, mas possui um campo que identifa uma requisição de monitoramento específica.

- **I am alive**: Esta mensagem é enviada por um QoSP/QoSPA/roteador para um QoSP em resposta à mensagem *Are you alive* (ou *Explicit ping*) previamente recebida. Ela equivale a um *Echo Reply* do protocolo ICMP. Tal mensagem possui dois campos cujos valores e tipos são os mesmos da mensagem *Are you alive* (ou *Explicit ping*) recebida.

Além das mensagens descritas acima, o QoSPM utiliza mensagens SNMP (estas não fazem parte do protocolo definido visto que o SNMP já é um protocolo padronizado) na sua comunicação. As mensagens SNMP são trocadas entre o QoSPA e o roteador vinculado ao mesmo, e tem por objetivo colher informações de QoS. As seguintes mensagens foram utilizadas:

- **Get Request**: Enviada por um QoSPA para o roteador vinculado ao mesmo quando o QoSPA deseja verificar se a QoS continua sendo mantida. Esta mensagem é utilizada para capturar o valor de uma ou mais variáveis MIB (descritas no capítulo 2).

- **Get Response**: Enviada por um roteador para o QoSPA no qual está vinculado em resposta à mensagem *Get Request* previamente recebida. Ela contém os valores das variáveis MIB requisitados.

5.4 ALGORITMOS DO QOSPM

Para completar o desenvolvimento do QoSPM, os algoritmos que o compõe foram especificados. Para cada algoritmo será descrita sua funcionalidade, assim como o contexto em que o mesmo executa. As mensagens citadas na descrição dos algoritmos são as mesmas descritas na seção anterior. Assim como na seção anterior, a palavra *QoSP* será utilizada de forma intercambiável com a expressão *módulo do QoSP* assim como a palavra *QoSPA* será utilizada de forma intercambiável com a expressão *módulo do QoSPA*.

- **Failure Detector** (Algoritmo 1): Algoritmo composto por duas tarefas que executam em paralelo no contexto do *Failure Detector* do QoSPM. Tais tarefas são responsáveis por detectar a falha dos componentes do QoSPM, mas especificadamente QoSP, QoSPA e roteadores. É importante ressaltar que cada QoSP monitora apenas os componentes que estão relacionados com os canais gerenciados pelo

mesmo. A tarefa *detectFailure* é responsável por enviar periodicamente mensagens *"Are you alive ?"* para os componentes de interesse do QoSP. A periodicidade do monitoramento é definido através do parâmetro *FD monitoring interval*. O vetor *elements* armazena os componentes que devem ser monitorados. A função *getElements()* retorna os elementos que devem ser monitorados ordenados pelo tempo de transferência de mensagens entre o QoSP local e o componente em questão, sendo que este tempo é calculado pela função *Delay*. A variável *ExpectedMsg* identifica o elemento cuja mensagem *"I am alive"* possivelmente será a próxima a ser recebida. Esta variável é compartilhada pelas duas tarefas. O vetor *ReplyStatus* guarda a informação dos elementos cujas mensagens *"I am alive"* já chegaram. Este vetor também é compartilhado pelas duas tarefas. A função *CT()* retorna a hora local. A declaração *Set EventWait(EVENT, timeout)* serve para indicar que a ocorrência de um evento será aguardada ou a ocorrência de um *timeout*, o que vier primeiro. Basicamente, o algoritmo funciona da seguinte maneira: baseado nas mensagens *"I am alive"* que ainda não chegaram, a função *EventWait* configura o *timeout* para o elemento cuja mensagem *"I am alive"* é a próxima esperada. Quando uma determinada mensagem não chegou no tempo esperado, é notificada a falha do componente correspondente. Uma vantagem deste algoritmo é que com apenas uma única *thread* de execução é possível monitorar todos os elementos concorrentemente, sendo que o tempo máximo para executar uma rodada deste algoritmo corresponde ao *delay* calculado para o último elemento do vetor *elements*.

A tarefa *fdListener* é responsável por ficar escutando o canal do *Failure Detector* e atualizando *ReplyStatus* com as informações dos elementos cujas mensagens *"I am alive"* já chegaram. Além do mais, ela é responsável por sinalizar a chegada da mensagem esperada.

Algoritmo 1: Failure Detector

```
1  Task detectFailure;
2  begin
3      At every FD monitoring interval do
4          elements ← getElements();
5          ExpectedMsg ← 1;
6          foreach element ∈ elements do
7              ReplyStatus[element] ← NotReceived;
8              AliveMsg.id ← element;
9              MsgSentTime[element] ← CT();
10             send "AliveMsg" message to element;
11         Set EventWait(ExpectedMsgReceived, Delay(localQoSP, elements[1]));
12         foreach element ∈ elements do
13             if ReplyStatus[element] = NotReceived then
14                 if element = ExpectedMsg then
15                     if element.type = QoSP then
16                         notifyQoSPFailure(element);
17                     else if element.type = QoSPA then
18                         notifyQoSPAFailure(element);
19                     else
20                         notifyRouterFailure(element);
21                     if element is the last then continue;
22                     ;
23                     ExpectedMsg ← ExpectedMsg + 1;
24                 else
25                     ExpectedMsg ← element;
26                 Set EventWait(ExpectedMsgReceived, MsgSentTime[ExpectedMsg] +
                     Delay(localQoSP, elements[ExpectedMsg]) − CT());
27 Task fdListener;
28 begin
29     wait for(receive message "AliveMsgReply");
30     ReplyStatus[AliveMsgReply.id] ← Received;
31     if ExpectedMsg = AliveMsgReply.id then
32         SignalEvent(ExpectedMsgReceived);
```

- **QoS** (Algoritmo 2): Responsável por verificar a QoS que está sendo provida ao canal de comunicação $c_{x/y}$ (equivale à função $QoS(p_x, p_y)$ descrita no capítulo 3). Esse algoritmo executa a tarefa *checkQoSP* em paralelo com o resto do algoritmo. A tarefa *checkQoSP* é responsável por verificar se um QoSP está vivo, através do envio da mensagem "*ExplicitPing*". Caso a resposta não chegue dentro do *timeout*, o resultado da verificação será *UNTIMELY*. Para verificar a QoS do canal $c_{x/y}$, o algoritmo verifica se $QoSP_y$ está vivo além de enviar mensagens "$MonitoringRequest$" para os QoSPA dos roteadores pertencentes ao canal $c_{x/y}$. Se todas as respostas dos QoSPA chegarem, informando que a QoS está mantida, o resultado da verificação é *TIMELY*, caso contrário será *UNTIMELY*. Observe que exceto pela tarefa *checkQoSP*, o algoritmo não estabelece *timeout* para o recebimento das mensagens.

Visto que os canais do QoS Provider usufruem dos serviços isócronos, caso as mensagens não cheguem, foi decorrente da falha de um componente do QoSPM, cuja falha será notificada pelo *Failure Detector*.

Algoritmo 2: QoS(p_x, p_y)

```
1  Task checkQoSP(QoSP_y);
2  begin
3      AliveMsg.id ← ExplicitPing;
4      send "AliveMsg" message to QoSP_y;
5      timeout ← CT() + Delay(localQoSP, QoSP_y);
6      wait for((receive message AliveMsgReply from QoSP_y) ∨ (CT() > timeout));
7      if CT () > timeout then
8          return UNTIMELY;
9  execute checkQoSP(QoSP_y);
10 foreach router ∈ channels[c_x/y] do
11     send "MonitoringRequest" message to QoSPA(router);
12 wait for(((receive message MonitoringReply(maintained) from QoSPA(router)) ∨
       (router ∨ QoSPA(router) has failed)) ∀router ∈ channels[c_x/y]);
13 if received every message MonitoringReply(maintained) ∧ ;
14 (∀message MonitoringReply(maintained), maintained = True) then
15     return TIMELY;
16 else
17     return UNTIMELY;
```

- **VerifyChannel** (Algoritmo 3): Equivale à função $VerifyChannel(c_{x/y})$ descrita no capítulo três. Caso o canal $c_{x/y}$ não seja gerenciado pelo QoSP local, uma requisição remota será executada atráves do envio da mensagem "$VerificationRequest$". O resultado da verificação remota será o resultado da função. Assim como no algoritmo $QoS(p_x, p_y)$, não utiliza-se *timeout*. Caso o canal seja gerenciado localmente, o resultado da função será encontrado através da execução da função *QoS*, seguido de uma verificação de tráfego no canal, se necessário. O resultado da função *VerifyChannel* será *TIMELY* se o resultado da execução da função *QoS* for *TIMELY* e logo em seguida for percebido que houve tráfego no canal $c_{x/y}$ durante um certo intervalo de tempo (estabelecido pelo modelo HA). Caso contrário o resultado da função será *UNTIMELY*.

Algoritmo 3: VerifyChannel($c_{x/y}$)

```
1  if c_{x/y} ∉ channels then
2      send "VerificationRequest(c_{x/y})" message to QoSP_x;
3      wait for((received message VerificationReply(c_{x/y}, qos) from QoSP_x) ∨ ;
4      (QoSP_x has failed));
5      if received message VerificationReply(c_{x/y}, qos) ∧ qos = TIMELY then
6          return TIMELY;
7      else
8          return UNTIMELY;
9  else
10     qos ← QoS(p_x, p_y);
11     if qos = UNTIMELY then
12         return UNTIMELY;
13     else
14         starttime ← CT();
15         timeout ← CHANNEL_DORMANCY_PERIOD;
16         wait for(CT() > timeout);
17         if lastMessageTime(p_x, p_y) < starttime then
18             return UNTIMELY;
19         else
20             return TIMELY;
```

- **notifyTimelyChannel** (Algoritmo 4): Representa a função que é chamada para notificar que a QoS de um canal ($c_{x/y}$) foi alterada para *TIMELY*. Informa aos QoSPA dos roteadores que compõem o canal $c_{x/y}$, para os quais este procedimento ainda não foi realizado, seu interesse em receber notificação de degradação. Além disso, registra os roteadores e o $QoSP_y$, ambos se necessário, junto ao *Failure Detector* para monitorá-los. Finaliza registrando o canal formado pelos processos p_x e p_y junto ao *Traffic Listener* (descrito na modelagem do QoSPM).

Algoritmo 4: notifyTimelyChannel(p_x, p_y)

```
1  foreach router ∈ channels[c_{x/y}] do
2      if QoSP_x is not subscribed with QoSPA(router) then
3          send "Subscribe" message to QoSPA(router);
4      if router is not being monitored by FailureDetector then
5          Set Failure Detector to monitor router;
6  if QoSP_y is not being monitored by FailureDetector then
7      Set Failure Detector to monitor QoSP_y;
8  register channels[c_{x/y}] with Traffic Listener;
```

- **notifyChannelsDegradation** (Algoritmo 5): Representa a função que é chamada para informar que um conjunto de canais teve sua QoS degradada para *UNTIMELY*. A degradação pode ter sido decorrente à notificação de falha (por parte de *Failure Detector*) de um QoSPA ou roteador ou de um QoSP, ou decorrente à notificação (por parte do QoSPA) de que um roteador não está podendo prover

um serviço previamente negociado, ou ainda decorrente à notificação (por parte do mecanismo de admissão) de que a QoS de um canal de comunicação foi rebaixada para *UNTIMELY*. Basicamente esse algoritmo realiza o procedimento inverso do algoritmo anterior, cancelando os registros dos elementos que não necessitam serem mais monitorados (em contextos diferentes, dependendo do elemento).

Algoritmo 5: notifyChannelsDegradation(downChannels)

```
1  foreach c_{x/y} ∈ downChannels do
2      channels_y ← {aux_{x/d} | (aux_{x/d} ∈ channels) ∧ (d = y)};
3      if {c_{x/y}} = channels_y then
4          unSet Failure Detector to monitor QoSP_y;
5      unRegister channels[c_{x/y}] with Traffic Listener;
6      notify p_x about c_{x/y} degradation;
7  foreach router managed by QoSP do
8      routedChannels ← {aux_{x/y} | (aux_{x/y} ∈ channels) ∧ (router ∈ aux_{x/y})} ;
9      if routedChannels ⊆ downChannels then
10         unSet Failure Detector to monitor router;
11         send "unSubscribe" message to QoSPA(router);
```

- **isQoSMaintained** (Algoritmo 6): Representa a função que será utilizada pelo QoSPA para verificar se a QoS contratada para os canais de comunicação *TIMELY* que passam pelo roteador monitorado pelo mesmo continua sendo provida. As mensagens "*GetRequest*" e "*GetResponse*" fazem do protocolo SNMP e são utilizadas para capturar o valor de uma variável MIB (*oid*). Quando observa-se que a QoS fornecida pelo roteador foi degradada, o QoSPA envia a mensagem "*Degradation*" para todos os QoSP que estão interessados em receber tal informação. Esta função também é chamada periodicamente para que o monitoramento automático do roteador possa ocorrer.

Algoritmo 6: isQoSMaintained()

```
1  send "GetRequest(oid)" message to router;
2  wait for((receive message GetResponse(oid) from router));
3  result ← ProcessOID(oid);
4  if result ≠ MAINTAINED then
5      foreach QoSP subscribed do
6          send "Degradation" message to QoSP;
7      return False;
8  else
9      return True;
```

As tarefas mencionadas nas descrições dos algoritmos (*detectFailure*, *fdListener* e *checkQoSP*) necessitam ser de tempo real para executar suas funções corretamente. Iste se deve ao fato das mesmas estabelecerem *timeout* para a execução de suas funções.

Capítulo 6

IMPLEMENTAÇÃO

Neste capítulo serão descritos alguns detalhes de implementação, além de descrever os testes realizados. A seção sobre os detalhes de implementação visa esclarecer como alguns problemas foram atacados, complementando as soluções abordadas no capítulo 5. A seção sobre testes abordará sobre em quais condições estes foram executados, os objetivos e os procedimentos adotados.

6.1 DETALHES DA IMPLEMENTAÇÃO

A seguir serão descritos alguns problemas e suas respectivas soluções que não foram abordados nas seções anteriores. Estes problemas não foram abordados anteriormente visto que suas soluções estão no nível da implementação.

6.1.1 Colhendo informações de QoS

Como já foi dito anteriormente, o protocolo utilizado para colher informações relativas a QoS nos roteadores foi o SNMP. Este último foi utilizado para colher informações relativas ao *Diffserv* configurado no roteador. Tais informações são estatísticas colhidas pelo agente do SNMP e armazenadas nas MIBs. Apesar de existirem MIBs proprietárias para colher informações relativas ao *Diffserv*, a IETF padronizou uma MIB para a arquitetura do *Diffserv* (BAKER; CHAN; SMITH, 2002).

Como em nosso ambiente de teste foi utilizado um roteador da Cisco modelo 871, a MIB utilizada para captura de informações relativas a QoS foi a MIB proprietária *CISCO-CLASS-BASED-QOS-MIB* (SYSTEMS, 2002) definida pela própria Cisco.

6.1.1.1 CISCO-CLASS-BASED-QOS-MIB Esta MIB provê acesso de leitura para informações de configuração e de estatísticas referente às classes de tráfego configuradas no roteador. As informações de configuração dessa MIB incluem todos os parâmetros relativos às classes, políticas e critérios de seleção configurados no roteador (para entender melhor estes conceitos e suas relações assim como a forma como estes são utilizados, veja o Apêndice A). As estatísticas disponíveis nesta MIB incluem contadores (tais como

quantidades de pacotes descartados) relativos às classes de tráfego antes e depois das políticas de QoS serem aplicadas. Uma das variáveis definidas nesta MIB é a seguinte:

- **cbQosCMDropPkt**: Este objeto MIB (cujo OID é 1.3.6.1.4.1.9.9.166.1.15.1.1.13) armazena o número de pacotes descartados por classe decorrentes a todas as características que causam descartes (como policiamento). Essas caracterísiticas são aquelas responsáveis por prover a funcionalidade necessária para implementar o *Diffserv*.

6.1.1.2 Descobrindo se a QoS está sendo degradada A degradação da QoS dos canais *TIMELY* é decorrente do mal fornecimento do serviço Expresso. Esse mal fornecimento pode ocorrer quando a classe que recebe o serviço Expresso está sobrecarregada. A sobrecarga faz com que componenetes do *Diffserv* como o policiamento e reguladores de fluxo (*Traffic Shaping*) causem descartes, compromentendo o serviço. Mesmo sabendo que o serviço Expresso conta com largura de banda garantida e que os roteadores de borda são responsáveis por garantir que as SLAs sejam respeitadas, em nosso ambiente de teste existe apenas um roteador, sendo assim nós não contamos com os roteadores de borda, havendo a possibilidade do serviço ser comprometido devido à sobrecarga.

Nós verificamos se houve degradação da seguinte maneira: o QoSPA envia duas mensagens SNMP, cada uma requisitando o valor do objeto MIB *cbQosCMDropPkt*. Calculamos a diferença entre os dois valores. Se a diferença for maior do que zero, pacotes estão sendo descartados e o serviço está comprometido (a QoS não está sendo mantida), caso contrário, a QoS dos canais que recebem tal serviço está sendo mantida. Como já foi comentado anteriormente, o QoSPA monitora periodicamente o roteador ao qual está vinculado (na verdade, assim que um monitoramento é finalizado, logo em seguida outro é executado). Sendo assim, cada vez que o monitoramento é executado, o QoSPA não necessita enviar duas mensagens SNMP requisitando o valor do objeto MIB *cbQosCMDropPkt*. Ele envia apenas uma mensagem e a diferença é calculada a partir do valor obtivo no monitoramento anteriormente executado. É importante observar que caso ambiente Diffserv esteja completo (composto por roteadores de borda e roteadores de núcleo), este tipo de degradação mencionado anteriormente não aconteceria, mas o SNMP poderia ser utilizado para colher valores de outros objetos MIB que também forneçam uma informação mais robusta acerca da degradação, como por exemplo verificar o corrompimento do *buffer*.

Para que as mensagens SNMP do tipo *SNMPGET* possam ser enviadas requisitando a quantidade de pacotes descartados da classe que recebe o serviço Expresso, antes é necessário que índices relativos à esta classe sejam capturados. Para saber como ocorre este procedimento, veja o Apêndice B.

6.1.2 Otimizando o monitoramento do QoSPA

Vários canais de comunicação *TIMELY* passam pelo mesmo roteador. Sendo assim, durante o processo de monitoração (executado através das funções *QOS* e *VerifyChannel*), vários QoSP podem requisitar dentro de um intervalo de tempo curto informações de QoS ao mesmo QoSPA. Além do mais, no processo de monitoração da QoS de um canal, o

monitoramento executado pelo QoSPA junto ao roteador possivelmente será um gargalo, visto que as mensagens do protocolo SNMP são maiores comparadas às mensagens do protocolo definido para o QoSPM, além do processamento executado pelo roteador ser mais lento se comparado aos *hosts*.

Para tentar otimizar o monitoramento realizado pelo QoSPA, duas medidas foram tomadas: o monitoramento é contínuo, sendo assim, quando uma solicitação por parte de um QoSP chega requisitando saber se a QoS continua sendo mantida, esta requisição se utilizará do monitoramento em execução, caso a resposta do roteador ainda não tenha chegado. Além do monitoramento ser contínuo, todas as requisições que chegam enquanto o monitoramento está sendo executado e, principalmente, se a resposta do roteador ainda não chegou, são armazenadas em um *buffer*. Ao fim do monitoramento, as requisições se utilizarão da resposta deste monitoramento.

6.1.3 Escutando os canais de comunicação

Para que a função $VerifyChannel(c_{x/y})$ (descrita no capítulo 3) possa ser implementada corretamente, é necessário que seja verificada a existência de tráfego no canal. Como foi descrito na seção que aborda a modelagem do QoSPM, a entidade *Traffic Listener* é responsável por esta função. Para que fosse possível armazenar as informações relativas à ocorrência de tráfego, duas funções foram definidas a serem utilizadas pelos processos aplicativos (que executam sobre o modelo HA) na hora de enviar e receber mensagens pelos seus canais de comunicação, criados junto ao QoS *Provider*. Tais funções são:

- *qosp_sendmsg*: utilizada para enviar uma mensagem pelo canal de comunicação. Ela é apenas um invólucro para as funções *sendmsg* e *rt_dev_sendmsg*. *sendmsg* faz parte da API do linux para programação em redes, sendo utilizada para envio de mensagens em um *socket*. *rt_dev_sendmsg* faz parte da API do RTnet, sendo utilizada para envio de mensagens em um *socket* de tempo real. Antes de executar uma das funções que estão sendo involucradas, *qosp_sendmsg* atualiza a informação da hora da última mensagem trocada no canal com a hora local.

- *qosp_recvmsg*: utilizada para receber uma mensagem pelo canal de comunicação. Ela é apenas um invólucro para as funções *recvmsg* e *rt_dev_recvmsg*. *recvmsg* também faz parte da API do linux para programação em redes, sendo utilizada para o recebimento de mensagens em um *socket*. *rt_dev_recvmsg* faz parte da API do RTnet, sendo utilizada para o recebimento de mensagens em um *socket* de tempo real. Depois de executar uma das funções que estão sendo involucradas, *qosp_recvmsg* atualiza a informação da hora da última mensagem trocada no canal com a hora local.

Cada função é um invólucro para duas funções, visto que dependendo da QoS do canal de comunicação utilizado pelo processo para enviar ou receber mensagem, a comunicação se dará no contexto de tempo real (API do RTNet) quando o canal for *TIMELY* ou no contexto do Linux quando for *UNTIMELY*. Ambas as funções definidas acima possuem tanto os mesmos parâmetros como o mesmo tipo de retorno das funções das quais elas

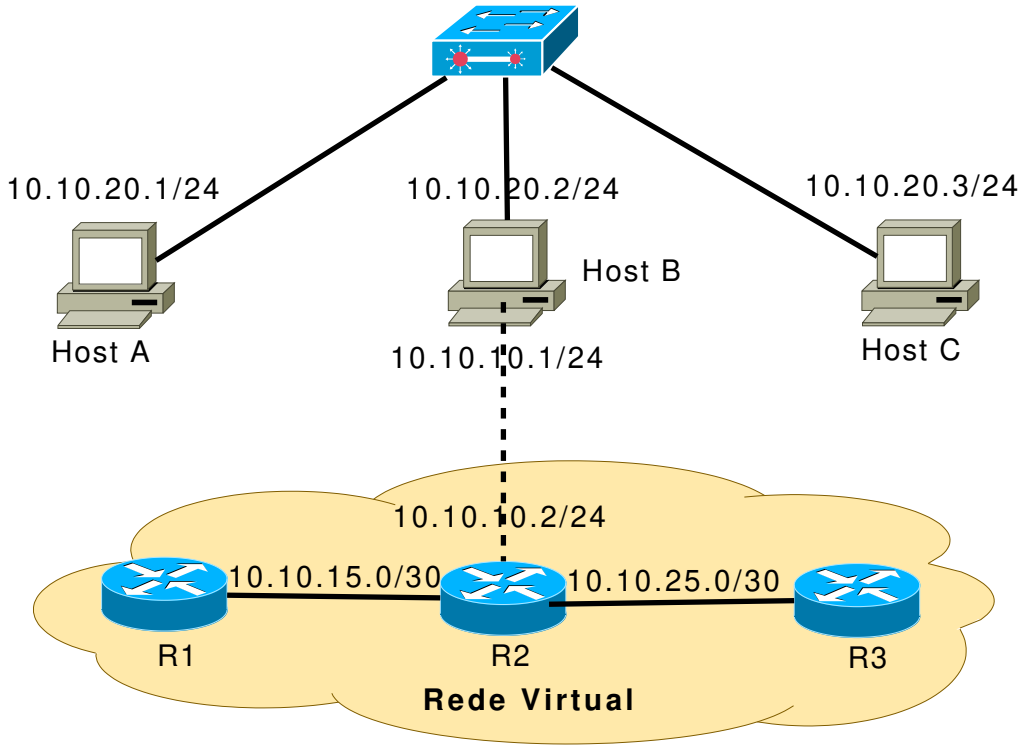

Figura 6.1 Ambiente de teste

são invólucros, exceto por um parâmetro adicional que corresponde a um descritor de canal. Para cada canal do processo aplicativo que teve sua QoS alterada para *TIMELY*, ele obtém um descritor para este canal que será utilizado como parâmetro nas funções de envio e recepção de mensagens por este canal, sendo que este parâmetro só será utilizado quando o canal for *TIMELY*.

6.2 TESTES

Devido às dificuldades encontradas (ver capítulo 7), medições não puderam ser realizadas para que resultados fossem colhidos. Alguns testes foram realizados na medida do possível. Testes de integração foram realizados para verificar se os componentes do QoSPM estavam se comunicando corretamente.

Para testar o QoSPM, nós utilizamos o laboratório localizado no módulo 2 do Lasid, pertencente ao projeto *Um Modelo Híbrido e Adaptativo Tolerante à Falhas*. O laboratório é composto por três computadores e um *switch* (Figura 6.1), sendo que uma rede local (10.10.20.0) foi configurada com os três computadores.

Existem três componentes QoSP executando na rede, um para cada *host*. Além disso, existe um componente QoSPA executando no *Host* B. O *Host* B servirá tanto como hospedeiro para um QoSP como para um QoSPA, visto que não haviam máquinas suficientes. Além de hospedar os componentes citados anteriormente, o *Host* B também passa a ser visto pelos componentes QoSP (inclusive pelo QoSP que executa em seu *host*) como o roteador da rede. É importante salientar que o *Host* B não se comporta como um roteador

(visto que o mesmo não rotea pacotes e apenas uma rede local foi configurada). O *Host* B é visto como roteador pelos QoSP para que os detectores de defeitos que executam nos mesmos possam ter um elemento a mais para monitorar.

O roteador que havia sido adquirido para o projeto (um modelo 871 da Cisco) acabou não pudendo ser utilizado (para maiores detalhes ver capítulo 7). Para testar o monitoramento dos roteadores através do SNMP foi utilizado o Dynamips/Dynagen (DYNAGEN,). O Dynamips permite emular uma rede composta por roteadores Cisco, além de outros elementos de rede como *switches*, em um computador. O Dynagen corresponde a um *front-end* para o Dynamips, permitindo a configução de um laboratório de rede facilmente. Com a utilização do Dynamips foi possível criar uma rede virtual (Figura 6.1) composta por três roteadores. O roteador R2 representa o roteador vinculado ao QoSPA, enquanto que R1 e R3 foram criados apenas para sobrecarregar R2. A rede virtual está "ligada"ao *Host* B através de R2 (observe que a rede virtual está hospedada no *Host* B e é vista apenas por ele), sendo que R2 era acessado através de uma interface virtual (10.10.10.1/30) criada com o *tunctl*. O *tunctl* é um comando utilizado para criar dispositivos TUN/TAP, sendo que estes dispositivos correspondem a *drivers* que implementam dispositivos de rede através de *software*, não necessitando de um adaptador de rede. TAP simula um dispositivo *Ethernet* e opera na camada 2, enquanto o TUN simula um dispositivo da camada de rede e opera na camada 3. É importante salientar que o roteador virtual criado é visto apenas pelo QoSPA e serve apenas para testar o monitoramento executado pelo mesmo. Este roteador criado fornece funcionalidades semelhantes ao modelo 871 da Cisco, inclusive dando suporte à CISCO-CLASS-BASED-QOS-MIB.

Para testar as funções *QoS* e *VerifyChannel* (ver capítulo 3), um canal de comunicação foi definido, sendo este formado pelo processo p_x que executa no *Host* A, e p_y que executa no *Host* B. Os processos ficam trocando mensagens entre si periodicamente. A função *VerifyChannel* foi executada no *Host* C para verificar o estado do canal $c_{x/y}$, sendo assim, tanto a função *VerifyChannel* como a própria função *QoS* foram testadas. Os valores retornados pela função *Delay* assim como o o intervalo de tempo durante o qual um canal deve ser investigado para verificar se houve tráfego foram configurados aleatoriamente, visto que o intuito era testar a integração entre os componentes.

Para testar se a degradação era detectada, os roteadores R1 e R3 foram criados para sobrecarregar o roteador R2. R1 ficava enviando periodicamente mensagens *ECHO REQUEST* para R3 enquanto que R3 ficava enviando periodicamente mensagens do mesmo tipo no sentido inverso. Os pacotes ICMP (*ECHO REQUEST*) foram configurados para receber o Serviço Expresso, sendo que este estava vinculado às interfaces 10.10.15.1 e 10.10.25.1 (na direção da saída) de R2. Quando o QoSPA requisitava informações da quantidade de pacotes descartados, sendo que esta requisição era feita no contexto da execução da função *QoS*, o QoSPA percebia a degradação e notificava a degradação.

Capítulo 7

CONCLUSÃO

O modelo proposto por (GORENDER; MACêDO; RAYNAL, 2007) mostrou-se inovador com relação ao fato de prover informações acerca dos estados dos processos em um sistema distribuído baseado na QoS disponível, sendo que o modelo adapta-se a esta QoS, permitindo que soluções eficientes sejam exploradas. Mas para que o modelo funcione corretamente, ele necessita gerenciar e acessar canais de comunicação com QoS. Este acesso é feito através do involúcro para arquiteturas de QoS denominado *QoS Provider*. Mais especificadamente, o modelo HA necessita de informações da QoS dos canais de comunicação para que os estados dos processos em um sistema distribuído possam ser atualizados corretamente.

Este trabalho visou a especificação e implementação do mecanismo de monitoramento do *QoS Provider*, responsável por prover informações sobre a QoS dos canais de comunicação gerenciados pelo *QoS Provider*. Procurou-se adotar princípios dos sistemas de monitoramento além de um modelo padrão para os sistemas de monitoramento.

A especificação do QoSPM se deu através da modelagem do QoSPM, descrição do protocolo utilizado pelo mesmo e o desenvolvimento dos principais algoritmos que o compõe. Na modelagem utilizou-se a abordagem orientada a objetos e foi feita através da UML. Quando necessário, houve modelagem tanto do domínio como da aplicação. O protocolo descreveu as mensagens trocadas entre os componentes do QoSPM (QoSP, QoSPA e roteadores). As principais tarefas executadas pelo QoSPM foram especificadas através de algoritmos. Além da especificação, detalhes acerda da implementação foram abordados.

Ainda com relação à especificação, uma arquitetura para o QoSPM foi especificada. A arquitetura do QoSPM é formada por dois componentes: módulos do QoSP e módulos do QoSPA. O QoSP é responsável por agregar informações referentes à QoS, sendo que estas foram colhidas junto aos roteadores pelos QoSPA, e também por detectar falhas dos componentes do QoSPM. No desenvolvimento da arquitetura, procurou-se adotar os princípios dos sistemas de monitoramento.

7.1 DIFICULDADES ENCONTRADAS

Inicialmente foram encontradas dificuldades com a configuração do ambiente, principalmente com relação à instalação do Xenomai. Cuidados com a configuração do arquivo

do *kernel* foram tomados para não comprometer a previsibilidade do Xenomai. Além disso, outro problema foi a falta do roteador que havia sido adquirido para o projeto. O roteador adquirido teve que ser devolvido visto que este não correspondia ao modelo que havia sido especificado e pelo qual havia sido pago, não sendo possível utilizá-lo para teste visto que não dava suporte ao *Diffserv*. Até o presente momento da elaboração desta monografia, o modelo correto ainda não havia chegado. Tivemos também problemas com as placas de rede que dão suporte ao RTnet, visto que as mesmas só foram adquiridas há pouco tempo. Outro problema encontrado foi a não conclusão dos outros módulos do *QoS Provider*, principalmente com relação ao módulo de cálculo do tempo de transmissão de mensagens em um canal de comunicação (corresponde à função *Delay* do *QoS Provider*). O detector de falhas que executa no componente QoSP do QoSPM necessita deste módulo para que os outros componentes do QoSPM possam ter suas falhas detectadas. A integração com o módulo de negociação não pôde ser realizada visto que este também não foi finalizado.

7.2 TRABALHOS FUTUROS

Os trabalhos futuros estão relacionados com a integração do QoSPM com os outros módulos do *QoS Provider*. A definição das estruturas de dados que devem ser compartilhadas entre os módulos assim como a forma como cada módulo deve utilizar a interface do outro faltam ser definidos. Testes completos devem ser realizados para que os resultados possam ser colhidos. Posteriormente à finalização do *QoS Provider*, este último deve ser utilizado pelos algoritmos do modelo HA a fim de verificar, principalmente, se os tempos são satisfatórios para acessar as informações de QoS.

REFERÊNCIAS BIBLIOGRÁFICAS

ASGARI, A. et al. A scalable real-time monitoring system for supporting traffic engineering. In: IEEE. *Workshop on IP Operations and Management*. [S.l.], 2002. p. 202–207.

AURRECOECHEA, C.; CAMPBELL, A. T.; HAUW, L. A survey of qos architectures. *In IEEE Transactions on Dependable and Secure Computing*, v. 4, n. 1, p. 18–31, 1996.

BAKER, F.; CHAN, K.; SMITH, A. *Management Information Base for the Differentiated Services Architecture*. [S.l.], 2002. RFC 3289.

BLAHA, M.; RUMBAUGH, J. *Modelagem e Projetos Baseados em Objetos com UML 2*. 1ª. ed. [S.l.]: Addison-Wesley, 2006.

BLAKE, S. et al. *An Architecture for Differentiated Services*. [S.l.], 1998. RFC 2475.

BOOCK, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language User Guide*. [S.l.]: Addison-Wesley, 1999.

CASE, J. et al. *A Simple Network Management Protocol (SNMP)*. [S.l.], 1990. RFC 1157.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Sistemas Distribuídos. Conceitos e Projeto*. 4ª. ed. [S.l.]: Pearson Education Limited, 2007.

DYNAGEN. Http://dynagen.org. Data de acesso: 2008.

EL-GENDY, M. A.; BOSE, A.; SHIN, K. G. Evolution of the internet qos and support for soft real-time applications. In: *Proceedings of the IEEE*. [S.l.: s.n.], 2003. v. 91, p. 983–985.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1995.

GERUM, P. *Xenomai - Implementing a RTOS emulation framework on GNU/Linux*. http://www.xenomai.org, 2004.

GODERIS, D. *Service Level Specification Semantics, Parameters and Negotiation Requirements*. [S.l.], 2001. IEEE Internet Draft: draft-tequila-sls-01.txt.

GORENDER, S. *Um Modelo Híbrido e Adaptativo para Sistemas Distribuídos Tolerantes a Falhas*. Tese (Doutorado) — Universidade Federal de Pernambuco, Março 2005.

GORENDER, S.; MACêDO, R.; RAYNAL, M. An adaptative programming model for fault-tolerant distributed computing. *In IEEE Transactions on Dependable and Secure Computing*, v. 4, n. 1, p. 18–31, 2007.

JIANG, Y.; THAM, C.; KO, C. Challenges and approaches in providing qos monitoring. *International Journal of Network Management*, v. 10, n. 6, p. 323–334, 2000.

KALINSKY, D. *Basic concepts of real-time operating systems*. http://www.kalinskyassociates.com, 2007. White Paper. Data de acesso: 2008.

KISZKA, J. et al. Rtnet - a flexible hard real-time networking framework. *In 10th IEEE Conference on Emerging Technologies and Factory Automation*, v. 1, 2005.

KLEITHY, J.; FRISBY, R.; ROGHLú, M. O. An initial investigation into qos provisioning. In: *Proceedings of Irish Telecommunications Systems Research Symposium (ITSRS)*. Dublin, Ireland: [s.n.], 2003.

MCKENNEY, P. *A realtime preemption overview*. http://rt.wiki.kernel.org, 2005. Data de acesso: 2008.

RTAI. Http://www.rtai.org. Data de acesso: 2008.

STALLINGS, W. *SNMP, SNMPv2 and RMON: Practical Network Management*. 2ª. ed. [S.l.]: Addison-Wesley, 1996.

SYSTEMS, I. C. *CISCO-CLASS-BASED-QOS-MIB*. 2002. Http://www.cisco.com. Data de acesso: 2008.

VERíSSIMO, P.; RODRIGUES, L. *Distributed Systems for Architectures*. 1ª. ed. [S.l.]: Kluwer Academic Publishers, 2001.

WANG, Z. *Internet QoS: Architecture and Mechanisms for Quality of Service*. 1ª. ed. [S.l.]: Morgan Kaufmann, 2001.

XENOMAI. Http://www.xenomai.org. Data de acesso: 2008.

XENOMAI. *A Tour of the Native API*. http://www.xenomai.org, 2006. Data de acesso: 2008.

XIAO, X.; NI, L. Internet qos: A big picture. *In IEEE Network*, v. 13, n. 2, p. 8–18, 1999.

YAGHMOUR, K. *Adaptive Domain Environment for Operating Systems*. http://www.xenomai.org, 2001.

YODAIKEN, V.; BARABANOV, M. A real-time linux. In: *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*. Anaheim, CA: [s.n.], 1997. The USENIX Association.

Apêndice

A

ENTENDENDO OS RELACIONAMENTOS EXISTENTES NA CISCO-CLASS-BASED-QOS-MIB

Para entender as características de QoS relacionadas às classes configuradas nos roteadores Cisco, é necessário compreender alguns conceitos relacionados aos objetos de QoS e seus relacionamentos existentes na MIB *CISCO-CLASS-BASED-QOS-MIB*. Objetos de QoS incluem mapas de classe (*ClassMaps*), padrões de comparação (*Match Statements*), mapas de políticas (*PolicyMaps*) e ações (*Feature Actions*).

- **Match Statement**: Critérios de casamento de padrão para identificar pacotes com o propósito de classificação.
- **ClassMap**: A definição de uma classe que contém vários *Match Statement* com o intuito de classificar pacotes em diversas categorias.
- **Feature Action**: Qualquer característica de QoS. Características de QoS incluem declarações (regras) de policiamento, marcação, políticas de enfileiramento, entre outras. Depois que um pacote é classificado, ele pode ter uma ação aplicada ao mesmo.
- **PolicyMap**: Uma política definida para associar *Feature Actions* às *ClassMaps*. Para que os serviços de QoS (como o *Diffserv*) possam ser oferecidos, uma *PolicyMap* deve ser vinculada a uma interface lógica de um roteador. Neste caso, a política passa a ser chamada de *Service Policy*.

Cada objeto de QoS possui uma informação de configuração e uma informação estatística (de instância) relativa ao mesmo. Tais informações são armazenadas em tabelas e são acessadas através de índices. Estes dois tipos de informações se diferem nos seguintes aspectos:

- **Informação de configuração**: Esta informação não muda mesmo se o objeto é vinculado à várias interfaces ou usado várias vezes. Ela é identificada unicamente para cada objeto com a mesma configuração através do índice *cbQosConfigIndex*.

- **Informação de instância**: Esta informação muda à medida que o objeto é vinculado à várias interfaces ou utilizado várias vezes. Cada uso de um objeto de QoS corresponde a uma instância deste objeto. Cada instância de um objeto é identificada unicamente em um dispositivo (roteador) através do índice *cbQosObjectsIndex*.

Além dos índices citados anteriormente, outro índice importante nesta MIB é o *cbQosPolicyIndex*. Este último é utilizado para identificar cada *PolicyMap* adicionada a uma interface. Para cada interface na qual está adicionada, uma *PolicyMap* terá um *cbQosPolicyIndex* diferente. Como o foco principal na análise da QoS está em obter informações relativas às políticas (*Service Policy*) e estas se relacionam com as outras características de QoS de forma hierárquica (para cada *PolicyMap* existem uma ou mais declarações *ClassMap*, sendo que para cada declaração *ClassMap* existe uma ou mais *Feature Action* associadas), os índices *cbQosPolicyIndex* e *cbQosObjectsIndex* são utilizados para identificar unicamente uma instância de objeto QoS do qual se interessa obter alguma informação. Cada vez que o roteador é reinicializado, os valores destes *indexes* são diferentes (ou melhor, possivelmente não serão iguais)

Todas as tabelas utilizadas na *CISCO-CLASS-BASED-QOS-MIB* relacionadas com informações de configuração serão acessadas através do *cbQosConfigIndex*, enquanto que para acessar as tabelas com informações estatísticas são necessários dois índice: *cbQosPolicyIndex* e *cbQosObjectsIndex*.

Apêndice

B

DESCOBRINDO OS ÍNDICES PARA ACESSAR INFORMAÇÕES ESTATÍSTICAS DAS CLASSES NA CISCO-CLASS-BASED-QOS-MIB

Como foi dito no Apêndice A (é necessário ler o Apêndice A para compreender este), as tabelas com informações estatísticas necessitam dos índices *cbQosPolicyIndex* e *cbQosObjectsIndex* para serem acessadas. O monitoramento dos roteadores será no nível da classe e, para que se possa verificar se a QoS nas classes está sendo mantida, informações estatísticas relativas a estas necessitam ser colhidas. A seguir será descrito o procedimento necessário para descobrir os dois índices citados anteriormente relativos à classe do serviço Expresso configurada no roteador. Considere que a seguinte *PolicyMap* está configurada no roteador com o intuito de tratar os fluxos dos canais *TIMELY*:

Comando 1:

```
class-map match-all premium
    match ip dscp 46
```

Comando 2:

```
policy-map HANDLE_TIMELY
    class premium
        priority 500
```

Estes comandos são utilizados para configurar o serviço expresso em um roteador Cisco. O primeiro comando é utilizado para classificar os pacotes que irão receber o serviço Expresso. Pacotes com o DSCP 46 receberão tal serviço. Observe que não foi mostrado como os pacotes foram marcados com este DSCP, caso eles realmente tenham sido marcados no roteador (isto está fora do escopo deste apêndice). O segundo comando configura uma política para tratar os pacotes que receberão o serviço Expresso. Tal política deve estar vinculada a uma interface (*Service Policy*) para que o serviço possa ser oferecido. Esta política deve ser aplicada à saída da interface e, dependendo do

sentido dos fluxos de tráfego e do número de redes que o rotedor interconecta, esta política deverá ser aplicada à várias interfaces. É importante salientar que os passos a seguir são utilizados para identificar os dois índices de interesse supondo que a política está sendo aplicada à apenas uma interface. Pequenas alterações necessitam ser feitas para levar em consideração mais de uma interface.

Antes de descrever os passos, é necessário falar um pouco da MIB *IP-MIB*. Ela é padronizada pelo IETF, sendo utilizada para gerenciar as implementações IP e ICMP nos elementos de rede. Esta MIB define o objeto MIB *ipAddEntIfIndex* (cujo OID é 1.3.6.1.2.1.4.20.1.2) responsável por identificar unicamente uma interface em um elemento de rede. Este identificador é também conhecido como *ifIndex*.

1. Obter o *ifIndex* que identifica unicamente a interface na qual está vinculada a política *HANDLE_TIMELY*

   Suponha que o IP de tal interface seja 10.10.10.2. Então, através do envio de uma mensagem *SNMP_GET* com o OID 1.3.6.1.2.1.4.20.1.2.10.10.10.2, nós conseguimos obter o *index* desejado. Vamos supor que tenha sido 1.

2. Obter o *cbQosPolicyIndex* da política *HANDLE_TIMELY* vinculada à interface em questão

   Dado que nós temos o *ifIndex* da interface de interesse, podemos descobrir as políticas vinculadas a esta interface através do objeto MIB *cbQoSIfIndex* (1.3.6.1.4.1.9.9.166.1.1.1.1.4), definido na MIB *CISCO-CLASS-BASED-QOS-MIB*. Este objeto faz parte de uma tabela (1.3.6.1.4.1.9.9.166.1.1.1) que descreve as interfaces e políticas vinculadas às mesmas. Tal objeto MIB retorna o *ifIndex* da interface para uma dada política na qual está vinculada. Como nós não temos ainda o *cbQosPolicyIndex* da política de interesse, vamos caminhar na árvore dos objetos MIB através do caminho identificado pelo OID 1.3.6.1.4.1.9.9.166.1.1.1.1.4 para descobrir todos os pares política-interface do roteador. Suponha que o resultado seja:

   1.3.6.1.4.1.9.9.166.1.1.1.1.4.1043 = INTEGER: 1

   1.3.6.1.4.1.9.9.166.1.1.1.1.4.1065 = INTEGER: 2

   Observe que no retorno da consulta, foi adicionado um número (a porção sublinhada) ao OID cuja árvore foi caminhada. Este número é exatamente o *cbQosPolicyIndex* que faltava para identificarmos a política. A segunda parte das igualdades identifica o *ifIndex* da interface. Como sabemos que *ifIndex* da interface em questão é 1, o *cbQosPolicyIndex* da política que nos interessa é 1043. No exemplo mostrado, existe apenas uma política vinculada à interface de interesse, mas poderiam existir duas políticas vinculadas, uma à saída e outra à entrada. Neste caso precisaríamos executar o passo seguinte.

3. Descobrir o *cbQoSPolicyIndex* da política vinculada à saída da interface

   Para executar este passo, vamos utilizar o objeto MIB *cbQoSPolicyDirection* (cujo OID é 1.3.6.1.4.1.9.9.166.1.1.1.1.3), definido no mesmo contexto (tanto da MIB como da tabela) do objeto MIB utilizado no passo anterior. Tal objeto MIB indica

a direção do tráfego para o qual a política será aplicada. Também vamos caminhar na árvore dos objetos MIB, mas neste caso através do caminho identificado pelo OID 1.3.6.1.4.1.9.9.166.1.1.1.1.3 para descobrir todos os pares política-direção do roteador. Suponha que em relação ao exemplo anterior, foram adicionadas políticas às entradas da interface, e que o resultado da caminhada seja:

1.3.6.1.4.1.9.9.166.1.1.1.1.3.1043 = INTEGER: 2

1.3.6.1.4.1.9.9.166.1.1.1.1.3.1044 = INTEGER: 1

1.3.6.1.4.1.9.9.166.1.1.1.1.3.1065 = INTEGER: 2

1.3.6.1.4.1.9.9.166.1.1.1.1.3.1066 = INTEGER: 1

A segunda parte das igualdades acima identifica o sentido do fluxo para o qual a política é aplicada. O valor 1 significa que a política é aplicada à entrada da interface, enquanto que o valor 2 significa o oposto. Se este mesmo exemplo se aplicasse ao passo anterior , nós estaríamos em dúvida (com relação ao *cbQoSPolicyIndex*) entre 1043 e 1044, supondo que ambos se aplicassem à interface identificada pelo *ifIndex* 1. Mas agora saberíamos que o *cbQoSPolicyIndex* da política em questão aplicada à interface de interesse é 1043, visto que esta política está sendo aplicada à saída da interface.

4. Descobrir o *cbQosConfigIndex* da classe premium

   Para executar este passo, vamos utilizar o objeto MIB *cbQoSCMName* (cujo OID é 1.3.6.1.4.1.9.9.166.1.7.1.1.1), também definido na MIB *CISCO-CLASS-BASED-QOS-MIB*. Este objeto faz parte de uma tabela (1.3.6.1.4.1.9.9.166.1.7.1) que especifica informações de configuração das classes (*ClassMap*). Tal objeto MIB retorna o nome da classe, identificada pelo *cbQosConfigIndex*. Como nós não temos ainda este índice, vamos caminhar na árvore dos objetos MIB através do caminho identificado pelo OID 1.3.6.1.4.1.9.9.166.1.7.1.1.1 para descobrir todos os pares classe-nome_da_classe do roteador. Suponha que o resultado seja:

   1.3.6.1.4.1.9.9.166.1.7.1.1.1.1025 = STRING: class-default

   1.3.6.1.4.1.9.9.166.1.7.1.1.1.1029 = STRING: premium

   Observe que no retorno da consulta, foi adicionado um número (a parte sublinhada) ao OID cuja árvore foi caminhada. Este número é exatamente o *cbQosConfigIndex* que identifica as classes configuradas no roteador. A segunda parte da igualdada mostra os nomes das classes. Como o nome da classe que nos interessa é *premium*, então sabemos que o *cbQosConfigIndex* desta classe é 1029. Apesar de nós não termos configurado a classe *best-effort* anteriormente, por padrão, todo roteador possui tal classe para classificar os pacotes que não foram agregados nas outras classes.

5. Descobrir o *cbQosObjectsIndex* da classe premium configurada na política *HANDLE_TIMELY*, vinculada à interface em questão

Para executar este passo, vamos utilizar o objeto MIB *cbQoSConfigIndex* (cujo OID é 1.3.6.1.4.1.9.9.166.1.5.1.1.2), também definido na MIB *CISCO-CLASS-BASED-QOS-MIB*. Este objeto faz parte de uma tabela (1.3.6.1.4.1.9.9.166.1.5.1) responsável por especificar a hierarquia de todos os objetos de QoS. Tal objeto MIB retorna o *cbQoSConfigIndex* para um dado objeto (identificado pelos índices *cbQosPolicyIndex* e *cbQosObjectsIndex*). Como nós já temos o *cbQosPolicyIndex* da política que nos interessa (1043) mas não temos ainda o *cbQosObjectsIndex* da classe *premium* para esta política, vamos caminhar na árvore dos objetos MIB através do caminho identificado pelo OID 1.3.6.1.4.1.9.9.166.1.5.1.1.2.1043 para descobrir todos os pares objeto-index_de_configuracao. Suponha que o resultado seja:

1.3.6.1.4.1.9.9.166.1.5.1.1.2.1043.1043 = Gauge32: 1035

1.3.6.1.4.1.9.9.166.1.5.1.1.2.1043.1045 = Gauge32: 1029

1.3.6.1.4.1.9.9.166.1.5.1.1.2.1043.1047 = Gauge32: 1033

1.3.6.1.4.1.9.9.166.1.5.1.1.2.1043.1049 = Gauge32: 1037

1.3.6.1.4.1.9.9.166.1.5.1.1.2.1043.1051 = Gauge32: 1025

1.3.6.1.4.1.9.9.166.1.5.1.1.2.1043.1053 = Gauge32: 1027

Os valores adicionados aos OID correspondem aos *cbQosObjectsIndex* dos objetos de QoS. A segunda parte da igualdada mostra os *cbQoSConfigIndex*. Como o valor deste último para a classe que nos interessa é 1029, então sabemos que o *cbQosObjectsIndex* desta classe para o *Service Policy* em questão é 1045.

Através destes passos, nós conseguimos descobrir os dois índices necessários para acessar as informações estatísticas da classe que fornece o serviço Expresso (*premium*). Estas informações estão armazenadas na tabela *cbQoSCMStatsTable* (cujo OID é 1.3.6.1.4.1.9.9.166.1.15.1). É importante salientar que todos estes passos devem ser executados apenas uma vez, durante o período de inicialização do QoS *Provider*. Caso o roteador seja reinicializado, o processo também deverá ser repetido, visto que os valores dos índices possivelmente serão diferentes.