Especificação e implementação de um mecanismo de monitoramento para um provedor de QoS distribuído

Autor: Hugo Vinícius Vaz Braga
Orientador: Sérgio Gorender

Contexto e motivação

- Sistemas distribuídos: processos se comunicam e coordenam suas ações através de troca de mensagens
- Diferentes ambientes de execução
  - níveis de sincronização
- QoS (*Quality of Service*) negociada nem sempre pode ser entregue.
- Arquiteturas de QoS permitem que esta seja negociada dinamicamente
- Modelo HA
  - Modelo para sistemas distribuídos tolerantes a falhas, híbrido e dinâmico
  - Acessa informações de QoS em nome dos processos aplicativos
    - → QoS Provider

QoS Provider (QoSP)

- Fornece e gerencia canais de comunicação com QoS
- Invólucro para arquiteturas de QoS
- Definido através de uma interface padronizada
  - Criação de canal
  - Negociação de QoS
  - Monitoração do canal
  - Cálculo de tempo para transferências de mensagens
- Dois tipos de canais de comunicação gerenciados
  - *Timely*
  - *Untimely*

Arquiteturas de QoS

- Host
  - Xenomai
    - *Framework* de desenvolvimento de tempo real
    - Executa de forma cooperativa com o kernel do Linux
  - Rtnet
    - *Framework* para comunicação de tempo real sobre Ethernet
    - Pilha de protocolos com restrições de tempo
- Roteador
  - Diffserv
    - Classes de comportamento

Colhendo informações de QoS: SNMP

- Protocolo padrão para gerência de redes
- Informações gerenciáveis são definidas através de variáveis listadas nas MIBs
- Utilizado para se obter uma informação mais robusta acerca da QoS

Mecanismo de monitoramento (QoSPM)

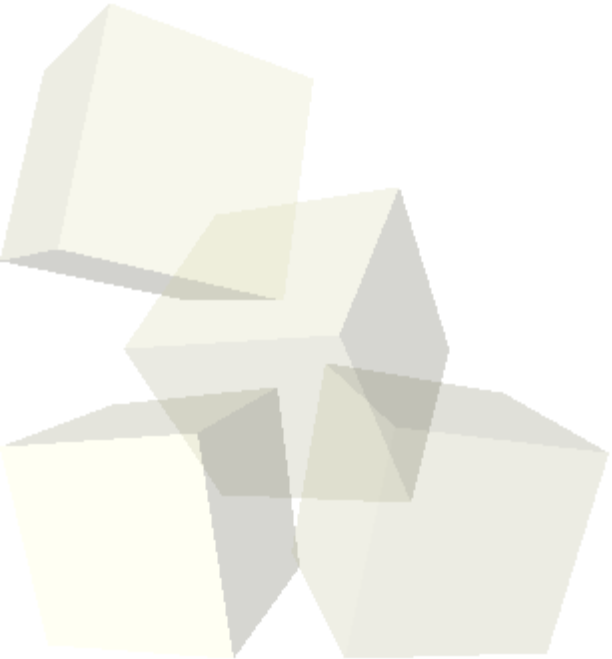
- Responsável pelo serviço de monitoração do QoSP
- Provê *feedback* sobre o estado atual da QoS
- Engloba as seguintes funções:
  - QoS(px, py): $\Pi^2 \rightarrow$ {*timely*, *untimely*}
  - VerifyChannel($c_{x/y}$): $\Gamma \rightarrow$ {*timely*, *untimely*}

Suporte ao QoSPM

- Princípios
  - Definir granularidade do processo de monitoramento
  - Distribuir o sistema coletor de dados
  - Minimizar o *overhead* da transmissão das medições
  - Adotar a política de notificação de eventos
- Modelo
  - Foca a distribuição do sistema de monitoramento

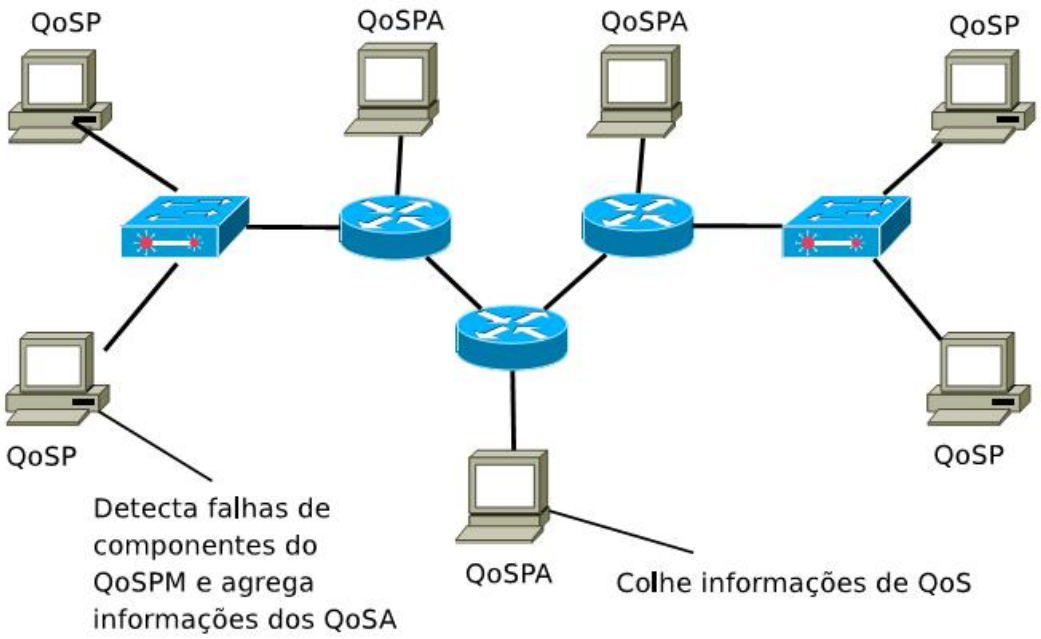
Especificação do QoSPM

- Arquitetura
- Modelagem UML
- Protocolo
- Algoritmos

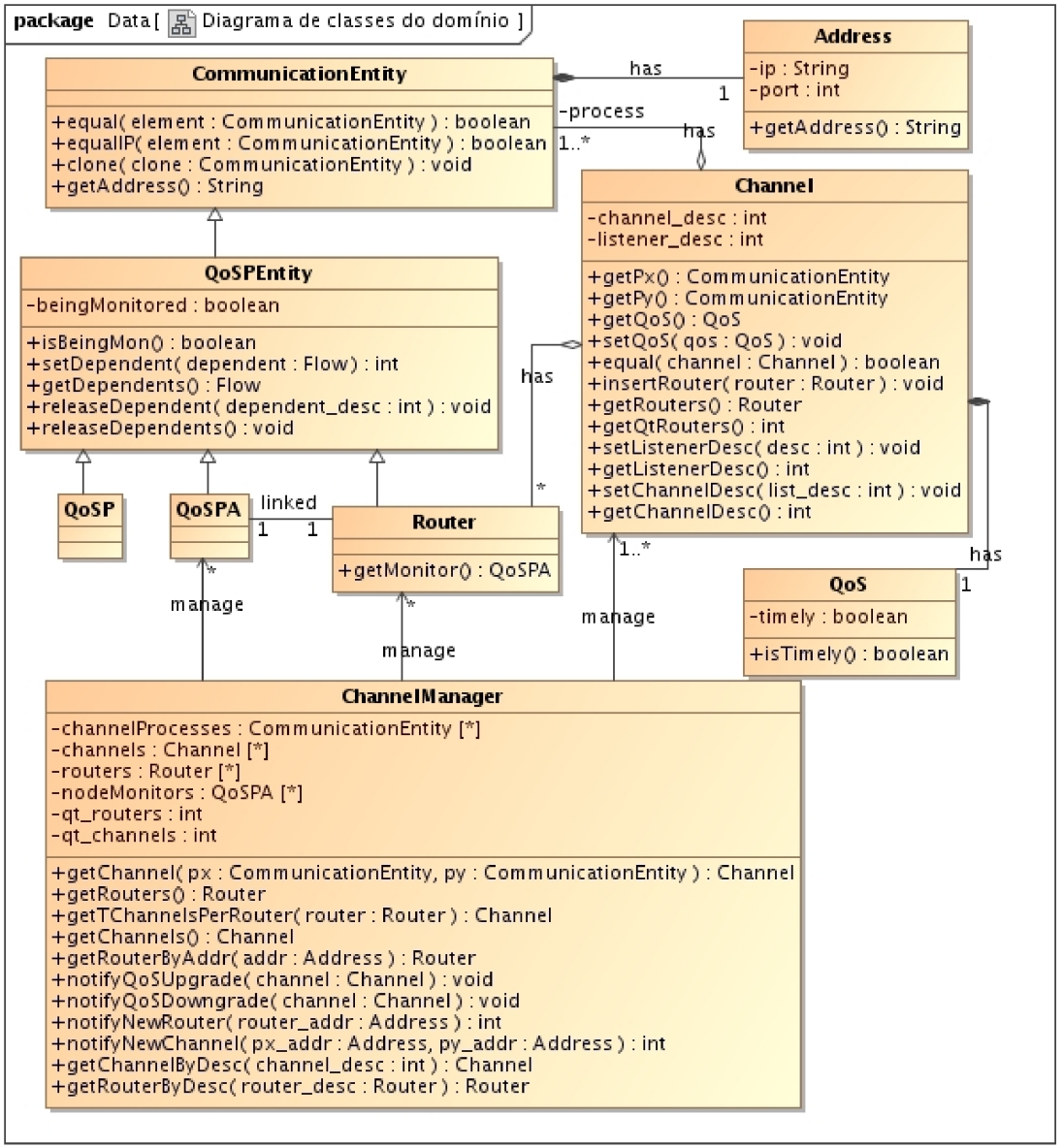


Arquitetura

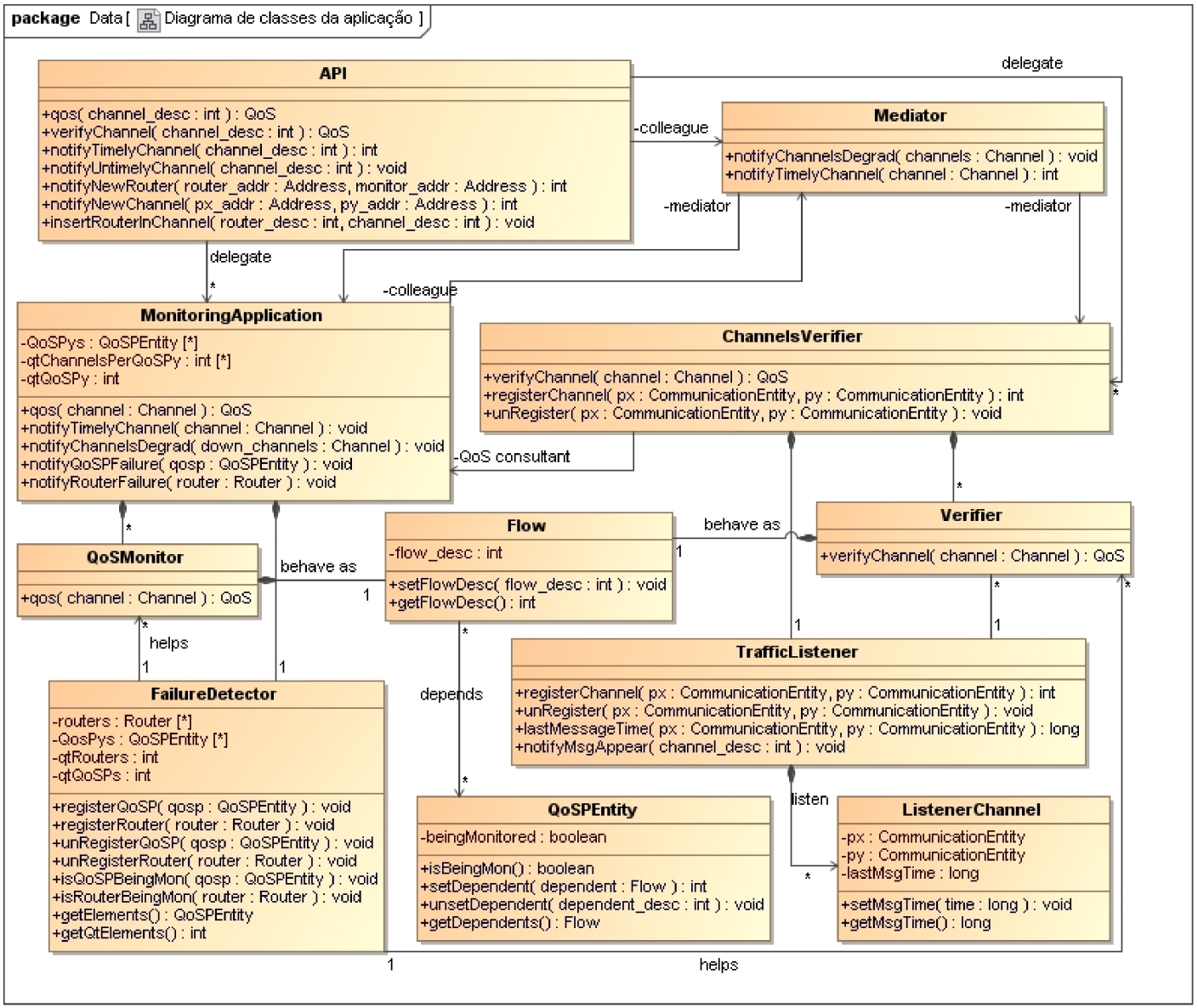
- Formada por dois componentes: QoSP e QoSPA (QoSP Agent)
- QoSP
  - Agrega informações de QoS colhidas pelos QoSPA
  - Detecta falhas de componentes do QoSPM
- QoSPA
  - Colhe informações de QoS através do SNMP
  - Notifica módulos do QoSP acerca da degradação da QoS



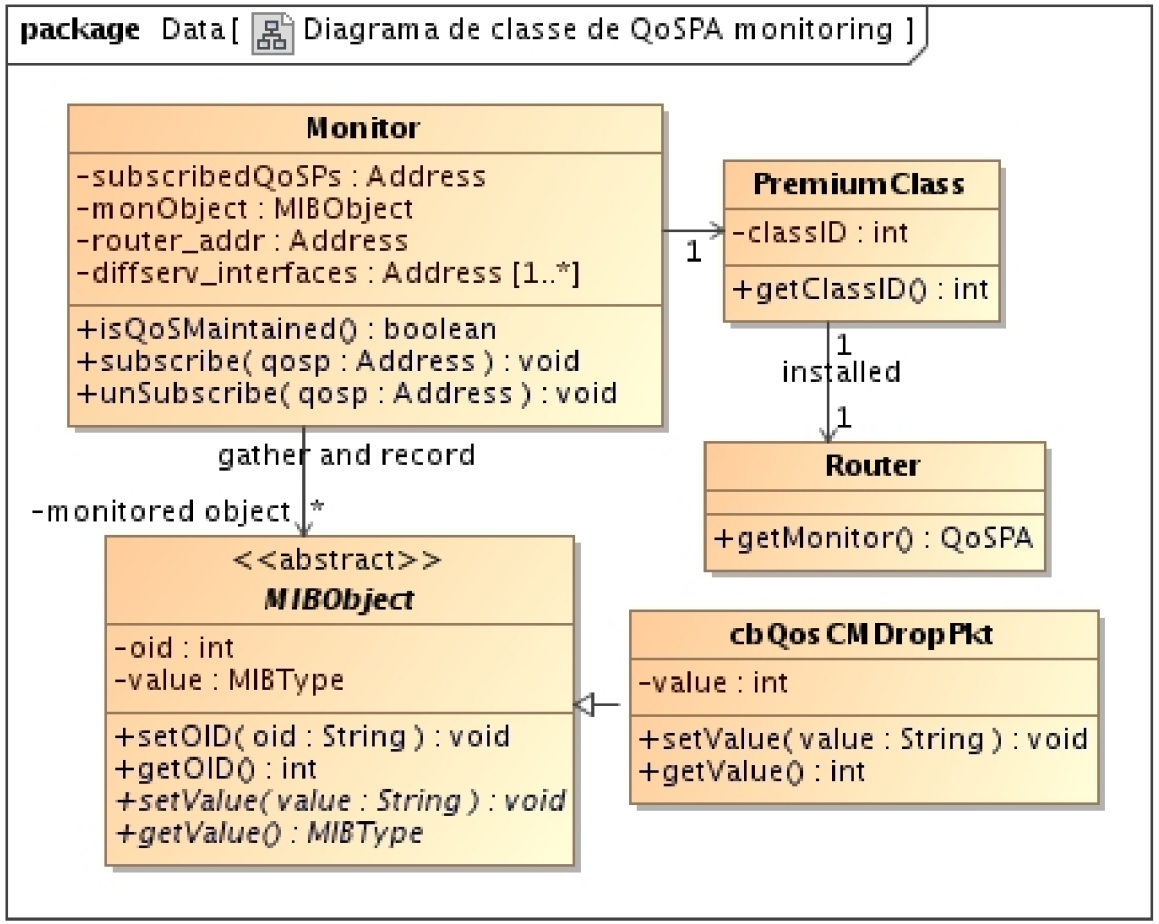
Modelagem do domínio



Modelagem do QoSP



Modelagem do QoSPA



Protocolo

- **Monitoring Request**
  - Origem: QoSP, Destino: QoSPA
  - Deseja saber se a QoS em um roteador está mantida
- **Monitoring Reply**
  - Origem: QoSPA, Destino: QoSP
  - Resposta à Monitoring Request informando se a QoS está mantida
- **Monitorin Reply Ack**.
  - Origem: QoSP, Destino: QoSPA
  - Confirmação de que a resposta do monitoramento chegou
- **Subscribe**
  - Origem: QoSP, Destino: QoSPA
  - Se inscreve para receber informações de degradação
- **Unsubscribe**
  - Origem: QoSP, Destino: QoSPA
  - Cancela a inscrição

Protocolo

- **Notify Degradation**
  - Origem: QoSPA, Destino: QoSP
  - Notificação sobre degradação
- **Verification Request**
  - Origem: QoSP, Destino: QoSP
  - Requisição remota de verificação de canal
- **Verification Reply**
  - Origem: QoSP, Destino: QoSP
  - Resposta da requisição remota de verificação de canal
- **Are you alive**
  - Origem: QoSP, Destino: QoSP/QoSPA/roteador
  - Verifica se um elemento está vivo
- **I am alive**
  - Origem: QoSP/QoSPA/roteador, Destino: QoSP
  - Informa que um elemento está vivo
- **Explicit Ping**
  - Origem: QoSP, Destino: QoSP
  - Verifica se um QoSP está vivo

Algoritmos: Failure Detector

```
Task detectFailure;
        At every FD monitoring interval do
                elements ← getElements();
                ExpectedMsg ← 1;
                foreach element ∈ elements do
                        ReplyStatus[element] ← NotReceived;
                        AliveMsg.id ← element;
                        MsgSentTime[element] ← CT();
                        send "AliveMsg" message to element;
                Set EventWait (ExpectedMsgReceived, Delay(localQoSP, elements[1]));
                foreach element ∈ elements do
                        if ReplyStatus[element] = NotReceived then
                                if element = ExpectedMsg then
                                        if element.type = QoSP then
                                                notifyQoSPFailure(element);
                                        else if element.type = QoSPA then
                                                notifyQoSPAFailure(element);
                                        else
                                                notifyRouterFailure(element);
                                        if element is the last then continue;
                                                ExpectedMsg ← ExpectedMsg + 1;
                                else
                                        ExpectedMsg ← element;
                                Set EventWait(ExpectedMsgReceived, MsgSentTime[ExpectedMsg] + Delay(localQoSP,
    elements[ExpectedMsg]) – CT());
Task fdListener;
        wait for (receive message "AliveMsgReply");
        ReplyStatus[AliveMsgReply.id] ← Received;
        if ExpectedMsg = AliveMsgReply.id then
                SignalEvent(ExpectedMsgReceived);
```

Algoritmos: QoS(p_x, p_y)

Task *checkQoSP(QoSP)*;
    AliveMsg.id ← *ExplicitPing*;
    send "*AliveMsg*" message to *QoSP*;
    timeout ← *CT*() + Delay(localQoSP,*QoSP*);
    wait for ((receive message *AliveMsgReply* from *QoSP*) ∨ (*CT*() > *timeout*));
    if *CT*() > *timeout* **then**
        return *UNTIMELY*;

execute *checkQoSP*($QoSP_y$);
foreach *router* ∈ *channels*[$c_{x/y}$] **do**
    send "*MonitoringRequest*" message to QoSPA(*router*);
wait for (((receive message *MonitoringReply*(*maintained*) from QoSPA(*router*)) ∨ (*router* ∨ QoSPA(*router*) has failed)) ∀*router* ∈ *channels*[$c_{x/y}$]);
if received every message *MonitoringReply*(*maintained*) ∧ (∀message *MonitoringReply*(*maintained*), *maintained* = True) **then**
    return *TIMELY*;
else
    return *UNTIMELY*;

Algoritmos: VerifyChannel($c_{x/y}$)

if $c_{x/y} \in$ *channels* **then**
    qos ← *QoS*(p_x, p_y);
    if qos = *UNTIMELY* **then**
        return *UNTIMELY*;
    else
        starttime ← *CT*();
        timeout ← *CHANNEL_DORMANCY_PERIOD;*
        wait for (*CT*(> *timeout*));
        if lastMessageTime(p_x, p_y) < *starttime* **then**
            return *UNTIMELY*;
        else
            return *TIMELY*;
else
    send "*VerificationRequest*($c_{x/y}$)" message to $QoSP_x$;
    wait for ((received message *VerificationReply*($c_{x/y}$, *qos*) from $QoSP_x$) ∨ ($QoSP_x$ has failed));
    if received message *VerificationReply*($c_{x/y}$, *qos*) ∧ *qos* = *TIMELY* **then**
        return *TIMELY*;
    else
        return *UNTIMELY*;

Algoritmos: alteração da QoS

notifyTimelyChannel(p_x , p_y):

foreach *router* ∈ *channels*[$c_{x/y}$] **do**
    if $QoSP_x$ is not *subscribed* with QoSPA(*router*) **then**
        send ”*Subscribe*” message to QoSPA(*router*);
    if *router* is not being monitored by *Failure Detector* **then**
        Set *Failure Detector* to monitor *router*;
if $QoSP_y$ is not being monitored by *Failure Detector* **then**
    Set *Failure Detector* to monitor $QoSP_y$;
register *channels*[$c_{x/y}$] with *Traffic Listener*;

notifyChannelsDegradation(*downChannels*):

foreach $c_{x/y}$ ∈ *downChannels* **do**
    $channels_y \leftarrow \{aux_{x/d} \mid aux_{x/d} \in channels \wedge (d = y)\}$;
    if $c_{x/y} = channels_y$ **then**
        unSet *Failure Detector* to monitor $QoSP_y$;
    unRegister *channels*[$c_{x/y}$] with *Traffic Listener*;
    notify p_x about $c_{x/y}$ degradation;
foreach *router* managed by *QoSP* **do**
    $routedChannels \leftarrow \{aux_{x/y} \mid aux_{x/y} \in channels \wedge router \in aux_{x/y}\}$;
    if $routedChannels \subseteq downChannels$ **then**
        unSet *Failure Detector* to monitor *router*;
        send ”*unSubscribe*” message to QoSPA(*router*);

Algoritmos: isQoSMaintained()

send “*GetRequest*(*oid*)” message to router;
wait for ((receive message *GetResponse*(*oid*) from router));
result ← *ProcessOID*(*oid*);
if *result* <> *MAINTAINED* then
    foreach *QoSP* subscribed **do**
        send “*Degradation*” message to *QoSP*;
    return *False*;
else
    return *True*;

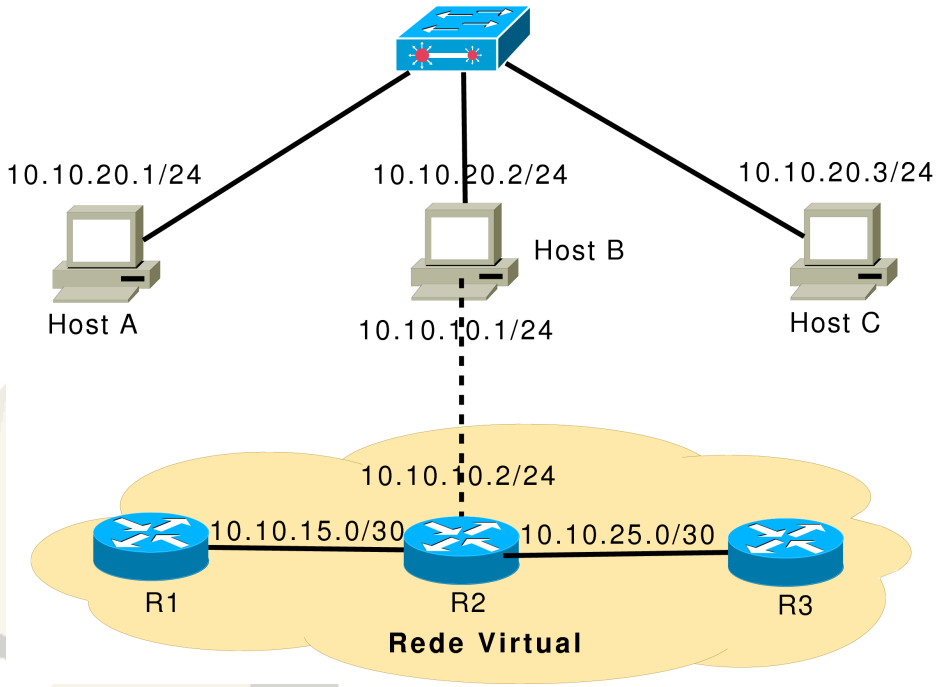
Detalhes da implementação

- Colhendo informações de QoS
  - Projeto baseado no uso de um roteador Cisco
  - Utilização da CISCO-CLASS-BASED-QOS-MIB
    - Utilizada para obter informações estatísticas referente às classes de tráfego
    - cbQosCMDropPkt
  - A degradação é percebida quando pacotes são descartados
- Otimizando o monitoramento do QoSPA
  - Monitoramento contínuo
  - Apenas um monitoramento para várias requisições

Detalhes da implementação

- Escutando os canais de comunicação
  - Necessário para que a função *VerifyChannel* possa ser implementada
  - Definida uma API para comunicação através dos canais gerenciados pelo QoS Provider:
    - qosp_sendmsg
    - qosp_recvmsg
  - API de comunicação é apenas um invólucro para a API do Rtnet e para a API do linux para programação em redes

Testes



- Um QoSP hospedado em cada *host*
- QoSPA hospedado no *host* B
- *Host* B hospeda também o roteador virtual, emulado com o Dynamips
- Dynamips permite emular uma rede com roteadores Cisco
- Canal criado entre *host* A e *host* B
- Função VerifyChannel executada em *host* C
- R1 e R3 sobrecarregaram R2 para simular a degradação

Conclusão

- Definição de uma arquitetura visando tornar o sistema escalável
- Especificação do mecanismo através da modelagem UML
- Desenvolvimento através da especificação dos algoritmos e de um protocolo utilizado pelo QoSPM
- Implementação na linguagem C
- Alguns testes de integração realizados

Dificuldades

- Configuração do Ambiente.
  - Instalação do Xenomai
- Problema com a falta do equipamento
  - Roteador
  - Placas de rede com suporte ao Rtnet
- Finalização dos outros módulos do QoS Provider

Trabalhos futuros

- Integração com os outros módulos do QoSP
- Utilização do mecanismo, como parte integrante do QoSP, pelos algoritmos do modelo HA

Bibliografia

- GORENDER, S.; MACÊDO, R.; RAYNAL, M. An adptative programming model for fault-tolerant distributed computing. In IEEE Transactions on Dependable and Secure Computing, v. 4, n.1, p. 18-31, 2007
- 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.
- XIAO, X.; NI, L. Internet QoS: A Big Picture. In IEEE Network, v. 13, n. 2, p. 8-18, 1999
- JIANG, Y.; THAM, C.; KO, C. Challenges and Approaches in Providing QoS Monitoring.
- A Scalable Real-time monitoring System for Supporting Traffic Engineering. In IEEE Proceedings. Workshop on IP Operations and Management, p. 202-207, 2002
- Xenomai. 2008. http://www.xenomai.org. Data de acesso: 2008