UNIVERSIDADE DO VALE DO RIO DOS SINOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM
COMPUTAÇÃO APLICADA

SIMMCAST como Ferramenta de Simulação para Avaliação de Protocolos de Roteamento Multicast

por

ROSANA CASAIS

Dissertação submetida a avaliação,
como requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof Dr. Antônio Marinho Pilla Barcellos
Orientador

São Leopoldo, novembro de 2002

"Este trabalho é dedicado a todas as pessoas que amo..."

Agradecimentos

Meus sinceros agradecimentos a todos que contribuiram para que este trabalho se tornasse realidade. Muitos foram co-responsáveis por este projeto.

Agradeço à minha família a compreensão pela ausência. À meus pais, por terem me ensinado o valor do conhecimento. Ao papai Odone, pela vida que partilhamos, e que cuidou como pai e mãe do bebê Luiza que nasceu durante este trabalho, sem descuidar da atenção destinada ao Lucas, que já fazia parte de nós. Agradeço, ainda, aos demais, por terem dedicado tempo quando precisei deixar tudo de lado para prosseguir o estudo.

Agradeço à empresa e aos colegas da Altus S. A. pela compreensão e apoio recebidos.

E, finalmente, agradeço a parceria obtida por professores e alunos do PIPCA, em particular ao bolsista de IC Hisham Muhammad pelo trabalho realizado no âmbito do roteamento estático, e ao meu orientador, Prof. Dr. Antônio Marinho Pilla Barcellos, pelo incentivo que recebi durante todo o percurso.

Este trabalho foi parcialmente financiado, primeiramente, pela empresa Altus S.A. e, após, pela Fundação de Amparo a Pesquisa do Rio Grande do Sul (FAPERGS), processo número 0060049.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API	Application Programming Interface
BGMP	Border Gateway Multicast Protocol
CBT	Core-Base Tree
DM	Dense Mode
DR	Designated Router
DVMRP	Distance Vector Multicast Routing Protocol
FIFO	First In First Out
GT-ITM	Georgia Tech Internetwork Topology Models
GUI	Graphical User Inteface
IGMP	Internet Group Management Protocol
MBGP	Multiprotocol Extensions for BGP-4
MOSPF	Multicast Extension for Open Shortest-Path First
MSDP	Multicast Source Distribuition Protocol
OO	Orientação a Objetos
PIM	Protocol Independent Multicast
POO	Programação Orientada a Objetos
QoS	Qualidade de Serviço
RBP	Response-Bucket Polling
RIP	Routing Information Protocol
RPF	Reverse Path Forwarding
RPT	RendezVous Path Tree
RP	RendezVous Point
SA	Sistema Autônomo
SM	Sparse Mode
SPT	Shortest Path Tree
TCP	Transmission Control Protocol
UML	Unified Modeling Language
VINT	Virtual InterNet Testbed

Resumo

Com o desenvolvimento das redes de computadores e o surgimento de sistemas com padrões mais complexos de comunicação, os simuladores tornaram-se ferramentas importantes e amplamente utilizadas para desenvolvimento, teste e avaliação de protocolos de comunicação em rede. As vantagens do uso de simuladores incluem a rapidez na geração de protótipos de protocolos e a possibilidade de criar cenários de simulação sem que seja necessário o uso de uma infra-estrutura física de rede. Este trabalho oferece o suporte necessário para simular protocolos de roteamento no ambiente de simulação de protocolos SIMMCAST. Inicialmente apresenta um estudo dos principais protocolos de roteamento intra e inter-domínios existentes na Internet, responsáveis pelo encaminhamento de pacotes em uma transmissão multicast, buscando apresentar uma abordagem comparativa entre as características necessárias para a realização de roteamentos eficientes. Em um segundo momento são exploradas as versões original da implementação do simulador de protocolos SIMMCAST, bem como a nova, incluindo capacidade de roteamento estático e dinâmico, este último com o desenvolvimento de protocolos de roteamento multicast. Na seqüência, são realizados testes e experimentos com a ferramenta SIMMCAST. Finalizando este trabalho, são apresentadas as conclusões, bem como possibilidades de desenvolvimentos futuros.

Palavras-chave: simulação de protocolos, roteamento, multicast.

TITLE: "ROUTING PROTOCOL EVALUATION THOUGH SIMMCAST"

Abstract

The development of computing networks and systems with complex communication patterns have being growing the importance of simulators. Today simulators are extensively used for the development, testing and evaluation of network communication protocols. Among the simulators benefits are the quick generation of protocol prototypes and the possibility of simulating network environments without the need to physically build them. This project provides the needed support to simulate routing protocols in the SIMMCAST protocol simulation environment. First it analyzes the main protocols to route intra and inter-domains existing in the Internet's the ones responsible to route the packs in a multicast transmission. This project part includes a comparison among the needed characteristics for efficient routing. The second part of the project presents the original architectures that implemented the SIMMCAST protocols simulator, as well as the new architecture with static and dynamic routing capabilities, this though multicast routing protocols develoment. Also presented the project covers testing and experiments performed by a SIMMCAST tool. Finally, the project conclusions and possibilities of future works.

Keywords: protocol simulation, routing, multicast.

Capítulo 1

INTRODUÇÃO

Multicasting é uma técnica que consiste no envio por um remetente de um mesmo conteúdo para múltiplos destinatários. Em uma transmissão multicast não há replicação desnecessária de pacotes. A utilização de multicast tem propiciado o uso abrangente de vários tipos de aplicações, tais como videoconferência e computação distribuída.

Multicast permite que aplicações escalem, ou seja, que sirvam a um grande número de usuários simultaneamente sem sobrecarregar a rede. Em aplicações com múltiplos participantes, a transmissão via multicast é vantajosa em relação à outras formas de transmissão; os pacotes são encaminhados a um conjunto de destinatários pretendido. Mais precisamente, o nível de rede garante a distribuição de pacotes para um grupo, para isso valendo-se de protocolos de roteamento multicast.

Roteadores multicast habilitam a distribuição eficiente de dados para múltiplos destinatários. Um protocolo de roteamento multicast deve ser escalável, robusto, utilizar o mínimo de recursos de memória, interagir com outros protocolos de roteamento e ser de fácil implementação ([35]). Roteamento multicast, especialmente inter-domínios, é um problema complexo envolvendo muitos aspectos como escalabilidade, políticas de roteamento e controle de acesso.

Escalabilidade é uma característica importante no roteamento multicast, que deve operar adequadamente em larga escala. Da mesma forma, concentração de tráfego também é um parâmetro importante. Ainda, a falta de um mecanismo de balanceamento de carga pode levar a situações em que um *link* inter-domínio esteja muito saturado com tráfego multicast, enquanto outros estão sem uso.

O uso de simulação permite que protocolos sejam avaliados de acordo com um conjunto de métricas. A simulação tem sido uma ferramenta poderosa no processo de projetar e avaliar protocolos de comunicação, em particular tratando-se de sistemas de larga escala. Ela permite que o projetista do protocolo ajuste o nível de detalhe, atendo-se apenas aos recursos desejados ([30]).

Um destes simuladores é o SIMMCAST ([6]), ambiente dedicado ao projeto e avaliação de protocolos multicast, de modo a permitir que os projetos de protocolos ali simulados aproximem-se, ainda mais, da realidade a que serão submetidos posteriormente. No SIMMCAST, é apresentado aos usuários um modelo de simulação de protocolos

abstrato e de fácil entendimento, e então, sobre este, pode-se, gradualmente, ampliar o nível de detalhe da simulação.

O simulador SIMMCAST, baseado em Java, permite que os protocolos simulados sejam especificados utilizando um *framework multi-thread* orientado a objetos. SIMMCAST foi construído visando, desde o início, protocolos multicast. Por suportar múltiplas *threads*, permite a especificação de protocolos que são construídos a partir de componentes simples, que simplificam enormemente o projeto do protocolo.

Viabilizar a simulação de roteamento quando do projeto de um protocolo de roteamento é de grande valia. Possibilitar a execução de simulações mais complexas, com redes em topologias arbitrárias no SIMMCAST, permite um uso muito mais abrangente deste simulador, permitindo a execução de experimentos até então não suportados. Incluem-se aqui experimentos que necessitam uma topologia com roteamento subjacente, bem como a simulação e avaliação de protocolos de roteamento em si, onde a camada de rede é o alvo do estudo.

O suporte a roteamento expande as capacidades do ambiente de simulação, permitindo, entre outros, a utilização de topologias dinâmicas, viabilizando experimentos de modelagem de falhas de *hosts*, roteadores ou *links*, bem como dispor de diferentes lógicas de roteamento multicast intra e inter-domínios, como árvore baseada em fonte ou compartilhada.

Em função disto, este trabalho explora a extensibilidade da arquitetura do SIMMCAST, criando componentes derivados a partir dos componentes básicos providos pelo *framework*. Além disso, nesta nova arquitetura, dois protocolos de roteamento foram implementados, viabilizando experimentos antes não suportados.

O texto desta dissertação está organizado de forma a apresentar o assunto estudado de forma progressiva. Desta forma, o Capítulo 2, Roteamento Multicast, apresenta uma revisão bibliográfica dos principais protocolos de roteamento multicast intra e inter-domínios. O Capítulo 3, Simuladores de Redes, traz uma revisão bibliográfica relativa a ambientes de simulação de redes de computadores. O Capítulo 4, O Simulador SIMMCAST, toma como base a bibliografia existente referente ao SIMMCAST ([6, 44]) e a estende, oferecendo os diagramas de classes UML ([23]), bem como uma discussão sobre a interface do *framework* SIMMCAST. Já o Capítulo 5, Roteamento Multicast no SIMMCAST, expande a arquitetura já apresentada, incluindo as classes necessárias para a realização de simulações envolvendo roteamento multicast no simulador. O Capítulo 6, SIMMCAST com Roteamento Dinâmico, viabiliza roteamento dinâmico para o simulador através da implementação de protocolos de roteamento. O Capítulo 7, Experimentos, descreve e analisa uma série de experimentos que exploram a funcionalidade acrescentada ao SIMMCAST. Finalmente, o Capítulo 8, Conclusões, finaliza este trabalho resumindo o que foi feito e potenciais trabalhos futuros com o simulador SIMMCAST. Em adendo, o Anexo A oferece o código fonte das implementações dos protocolos.

Capítulo 2

ROTEAMENTO MULTICAST

Denomina-se roteamento o processo de mover pacotes de um módulo origem para um módulo destino, em uma mesma rede ou entre sub-redes, integrantes de uma interconexão de redes ([66]). A atividade de rotear necessita que tenha sido determinado o melhor caminho de roteamento. Para determinar o melhor caminho, algoritmos de roteamento utilizam uma métrica, a qual pode ser tamanho do caminho, confiabilidade, atraso, largura de banda, carga e custo de comunicação, por exemplo. São os roteadores os responsáveis por determinar as melhores rotas, de acordo com alguma métrica.

O foco deste trabalho é roteamento no SIMMCAST. Em função disto, este capítulo apresenta uma revisão bibliográfica relativa a roteamento multicast. A Seção 2.1 aborda estratégias de roteamento de pacotes. As Seções 2.2 e 2.3 trazem uma revisão dos principais protocolos de roteamento multicast intra e inter-domínios. Dois destes protocolos foram implementados e são descritos no Capítulo 6. Ainda, a Seção 2.4 apresenta características a serem consideradas quando se analisa desempenho de protocolos de roteamento.

2.1 Roteamento de Pacotes para Múltiplos Destinatários

Roteamento multicast permite que pacotes sejam distribuídos eficientemente até sub-redes e daí a *hosts* destinatários. Os protocolos de roteamento multicast possuem como objetivo principal "montar" a árvore multicast que será utilizada no encaminhamento dos dados. A informação sobre a topologia da árvore multicast fica armazenada em roteadores, e indica para que interfaces um pacote de um fonte S destinado a um grupo G deve ser encaminhado.

Muitos algoritmos de roteamento multicast foram propostos e, a partir destes algoritmos foram projetados e implementados diversos protocolos de roteamento multicast intra-domínio. São chamados intra-domínio os protocolos executados por roteadores de dentro de um determinado Sistema Autônomo (SA). Estes permitem que sejam definidas as rotas para dentro da rede de uma determinada organização.

Um problema ainda mais desafiador do que multicast em um domínio é o roteamento multicast na Internet como um todo, envolvendo múltiplos domínios. Desta

necessidade surgiram os protocolos de roteamento inter-domínios, executados por roteadores localizados nos limites dos domínios e que permitem a definição das rotas que são utilizadas para a comunicação com dispositivos de fora de um determinado SA.

Roteamentos eficientes podem ser medidos em termos de uma variedade de parâmetros, em particular atraso mínimo entre o transmissor / cada receptor e custo de rede global. No encaminhamento de pacotes a destinatários, o objetivo é entregar pacotes somente aos *hosts* que compõem o conjunto. Os seguintes métodos podem ser utilizados para isto ([5]):

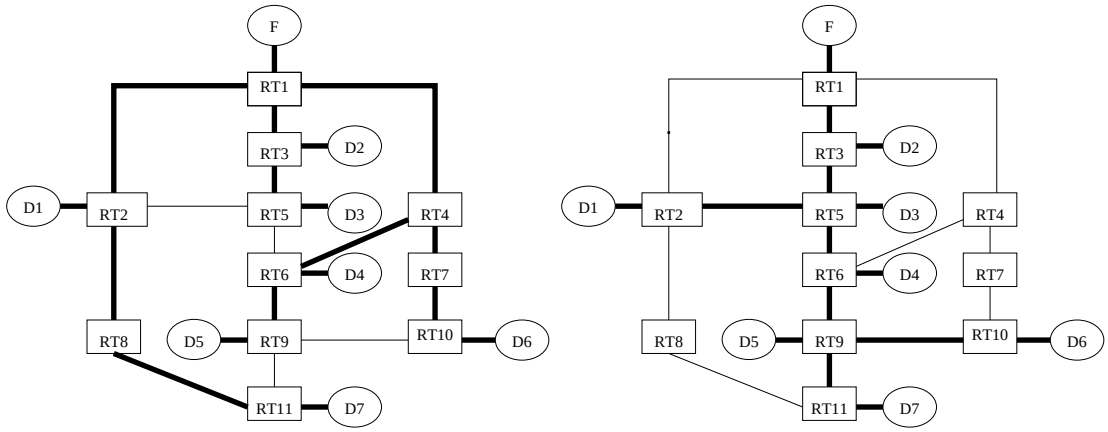
- múltiplos unicasts: de posse da lista de endereços de rede dos destinatários, o fonte executa uma transmissão individual para cada um deles. Cada cópia do pacote que é enviada é idêntica às demais exceto pelo valor do endereço de rede destino no cabeçalho do pacote;
- endereçamento multidestino: envio de um pacote contendo não um destino, mas sim uma lista deles. Cada pacote é encaminhado de acordo com a atual lista de endereços no cabeçalho do pacote, e a medida que o pacote é distribuído, endereços vão sendo removidos do mesmo;
- inundação: o *host* fonte inicia enviando o pacote a todos os seus vizinhos. Cada roteador que recebe o pacote envia uma cópia a todos os seus vizinhos, menos àquele de quem o pacote foi recebido. O pacote se espalha tal e qual uma reação em cadeia;
- árvores de distribuição multicast: para a propagação de um pacote, é selecionado um subconjunto de roteadores da rede, formando uma estrutura em árvore. Na raiz da árvore está o fonte, nas folhas das árvores os destinatários, e os demais nós (nós internos) são roteadores. Estas árvores podem ser de dois tipos: árvore baseada no fonte[1] e árvore compartilhada[2], e são detalhadas a seguir.

Tanto na árvore baseada no fonte como na compartilhada, é necessário se determinar uma árvore de distribuição multicast ([39]). Os algoritmos mais comuns para a geração da árvore são:

- algoritmo Árvore de Caminho Mais Curto (SPT)[3], cujo objetivo é minimizar individualmente a distância entre cada destinatário e o fonte. Em geral, minimiza a latência entre cada destinatário e o fonte;
- algoritmo Árvore de Custo Mínimo[4], que visa reduzir o custo global de transmissão de pacotes.

A Figura 2.1 apresenta exemplos de árvore de caminho mais curto e custo mínimo. Na árvore de caminho mais curto, a distância máxima entre um destinatário qualquer e o fonte é 5 hops, enquanto que na árvore de custo mínimo este valor é 7. No entanto, o número total de *hops* utilizados na árvore de caminho mais curto é 18, enquanto que na árvore de custo mínimo é 15 ([48]).

[1] source-based tree
[2] shared tree
[3] shortest path tree / Bellman-Ford - Dijkstra
[4] minimum spanning tree / Árvore Mínima de Steiner



(a) caminho mais curto (b) custo mínimo

FIGURA 2.1 – Árvore de Caminho Mais Curto x Árvore de Custo Mínimo

2.1.1 Árvore de Distribuição Baseada no Fonte

Nas árvores de distribuição baseadas no fonte, idealmente, cópias do pacote multicast devem ser confinadas às partes da rede que contém *hosts* destinatários. Ou seja, cópias do pacote podem ser roteadas apenas às subredes contendo *hosts* destinatários.

Adicionalmente, cada cópia do pacote deve atravessar uma única vez cada *link*, sendo o pacote replicado quando necessário. Para propagação do pacote, é selecionado um subconjunto de roteadores da rede, formando uma estrutura em árvore. Na raiz da árvore está o fonte, nas pontas da árvore estão os destinatários e os demais nós (internos) são roteadores.

A Figura 2.2 ilustra uma rede onde as rotas de uma árvore multicast do fonte aos destinatários está realçada, sendo que os roteadores RT7 e RT8 não fazem parte da árvore.

2.1.2 Árvore de Distribuição Compartilhada

A principal característica da árvore compartilhada é possuir um núcleo. Em geral, o pacote a ser distribuído é primeiro enviado ao núcleo pelo fonte; o roteador núcleo então redistribui uma cópia do pacote para cada destinatário, segundo uma árvore multicast cuja raiz é o núcleo. Quanto há múltiplos fontes enviando para um grupo, eles compartilham uma única árvore de distribuição multicast.

Em árvores baseadas no fonte há uma árvore multicast para cada par (fonte, grupo), enquanto que com árvores compartilhadas há uma única árvore por grupo (independente do número de fontes). Porque exigem menos informações de estado em roteadores, as árvores compartilhadas são mais escaláveis do que as árvores baseadas no fonte.

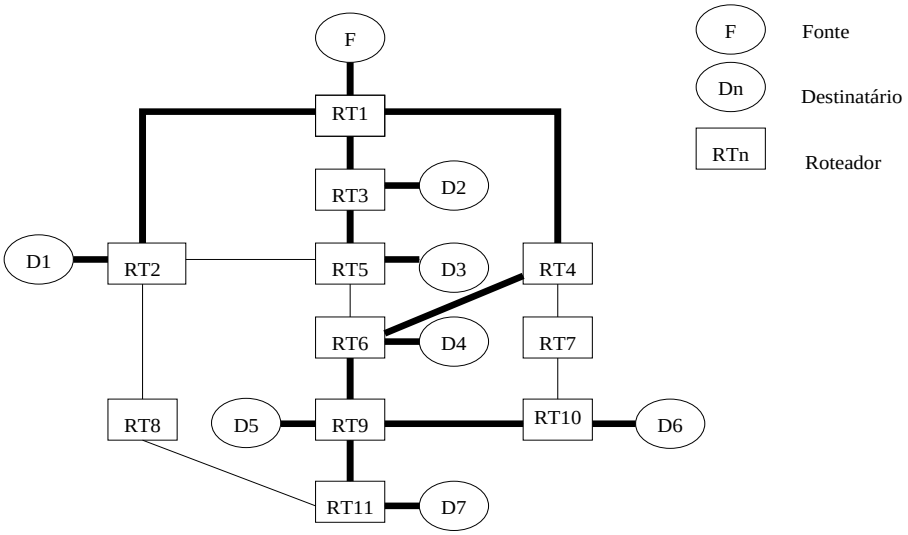


FIGURA 2.2 – Árvore Multicast Baseada no Fonte

A Figura 2.3 ilustra uma configuração onde há dois fontes, F1 e F2, e 6 destinatários, D2 a D7. Os fontes não fazem parte do grupo multicast, e o roteador núcleo é o RT5, que propaga os pacotes aos destinatários de acordo com a árvore realçada na figura.

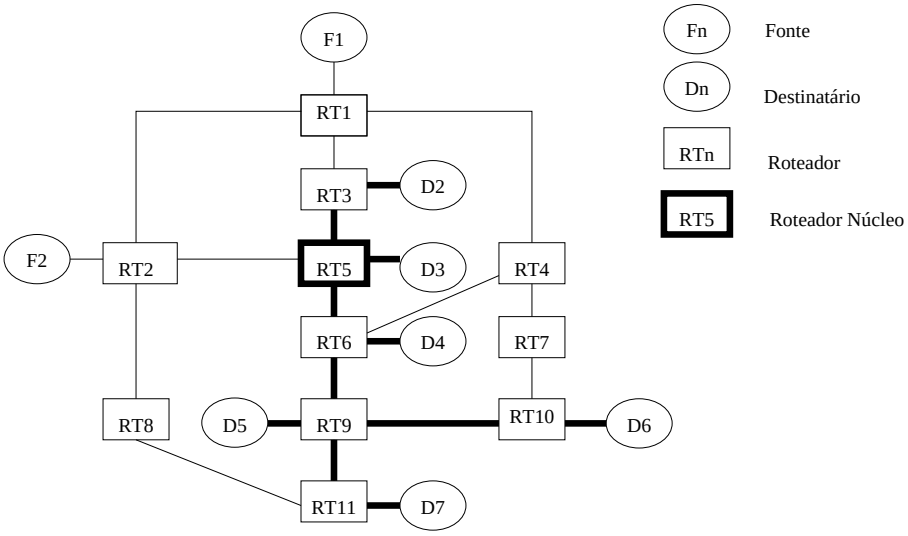


FIGURA 2.3 – Árvore Multicast Compartilhada

2.2 Protocolos Intra-Domínio

Protocolos de roteamento multicast são executados por roteadores multicast. Roteadores multicast ligados a uma ou mais sub-redes executam, ainda, um protocolo de gerenciamento de grupo, denominado *Internet Group Management Protocol* - IGMP ([14]), que adiciona informações de grupo aos protocolos de roteamento multicast e é utilizado pelos nodos *host* para entrar ou sair de determinado grupo multicast.

O protocolo IGMP permite que os roteadores multicast aprendam quais grupos possuem membros em cada uma das suas interfaces, armazenando os grupos multicast que possuem *hosts* clientes. No IGMP, o roteador faz um broadcast periódico na subrede, consultando se há *hosts* com processos integrantes de qualquer grupo multicast. Qualquer *host* que tenha um ou mais membros responde à consulta enviando à subrede um relatório para cada grupo assinado. Com isto, o roteador pode coletar relatórios e determinar quais grupos estão sendo assinados na subrede que ele controla. Ao roteador não interessa saber quais *hosts* estão assinando quais grupos, mas apenas quais grupos. Existem três versões de IGMP (v.1, v.2, v.3), sendo IGMPv.2 a versão mais popularmente encontrada. Para maiores informações sobre IGMP, vide [21].

Como já mencionado, todos os roteadores multicast executam protocolos de roteamento multicast, sejam eles de borda ou não. Os principais protocolos de roteamento multicast na Internet, que permitem que pacotes sejam encaminhados do fonte aos múltiplos destinos dentro de um mesmo domínio, são:

- DVMRP (*Distance Vector Multicast Routing Protocol*);
- MOSPF (*Multicast Extension for Open Shortest Path First*);
- PIM/DM (*Protocol Independent Multicast / Dense Mode*);
- PIM/SM (*Protocol Independent Multicast / Sparse Mode*);
- CBT (*Core Base Tree*).

Estes protocolos são descritos nas próximas subseções.

2.2.1 Protocolo DVMRP

É um protocolo intra-domínio baseado em vetor de distâncias. Os pacotes são inundados por toda a rede de acordo com uma árvore reversa de menor caminho. Uma característica do DVMRP é a contenção de tráfego multicast através de um esquema de *graft*. A árvore resultante não possui ramos onde não existam um ou mais *hosts* assinantes, evitando transmissões desnecessárias. A seguinte metodologia de operação é adotada ([62]):

1. A origem manda os pacotes para sua rede. Um roteador ao qual ela esteja ligada recebe e manda os pacotes por todas as suas saídas.

2. Todos os roteadores que recebem um pacote fazem uma checagem de RPF (*Reverse Path Forwarding* ([12])), isto é, o roteador verifica se a interface por onde veio o pacote é a interface que ele usaria para mandar pacotes para a origem. Se sim, o pacote é transmitido para as outras interfaces. Se não, o pacote é descartado.

3. Quando chega a um roteador com *hosts* ligados a ele, o roteador verifica se existe algum *host* que é membro do grupo destino do pacote, utilizandos testes periódicos de IGMP. Se há membros do grupo, o pacote é transmitido para a sub-rede. Se não, uma mensagem *prune* é transmitida de volta pela interface RPF, informando que não há membros do grupo naquela sub-rede.

4. Se um roteador recebe pacotes *prune* de todas as suas interfaces menos a RPF, ele também manda uma mensagem *prune* pela RPF. Como o DVMRP foi desenvolvido para rotear tráfegos multicast e não unicast, um roteador precisa executar múltiplos processos de roteamento: um para rotear o tráfego unicast e outro para o tráfego multicast. O processo DVMRP troca periodicamente mensagens de atualização de tabelas de roteamento com seus roteadores vizinhos capazes de lidar com tráfego multicast. Estas atualizações são independentes daquelas geradas por qualquer protocolo de roteamento unicast.

2.2.2 Protocolo MOSPF

MOSPF é um protocolo de roteamento que estende o protocolo unicast OSPF ([43]). Como tal, está baseado no estado do *link*: cada roteador mantém uma base de dados de nodos e *links* que descreve a topologia. Quando há uma alteração, um roteador envia, por inundação, um aviso a todos os demais roteadores. O MOSPF aplica o mesmo conceito para multicast, fazendo com que roteadores armazenem informações de composição de grupo: quando um novo grupo é assinado, o roteador inunda a rede com a informação. De posse da topologia e da informação de grupo, cada roteador controi sua árvore de distribuição multicast.

A árvore de distribuição é calculada através do algoritmo Dijkstra ([27]). É importante notar que o pacote multicast é enviado através de uma árvore de menores caminhos. E, estes caminhos, dependem dos endereços origem e destino, chamado roteamento origem/destino, diferindo da maioria dos outros protocolos de roteamento, baseados apenas no endereço destino.

Entretanto, a otimização em termos de escolha dos menores caminhos de uma origem para os componentes de um dado grupo não significa otimização por completo nas sub-redes. Para isto devem ser utilizadas árvores de cobertura mínima[5], como alternativa.

2.2.3 Protocolos PIM

Os protocolos estudados até agora tiveram como base protocolos de roteamento unicast: DVMRP surgiu a partir do RIP ([48]) e MOSPF é derivado do OSPF (*Open Shortest Path First*). O PIM ([15, 17, 18]) foi concebido de forma a não depender dos protocolos de roteamento unicast já existentes. Pode ser dividido em dois protocolos diferentes, conforme a distribuição do grupo pela rede:

- quando a disposição do grupo multicast apresentar uma alta densidade, o PIM/DM (PIM - *Dense Mode*) é utilizado;
- quando o grupo encontra-se disperso pela rede, o mais indicado é o PIM/SM (PIM - *Sparse Mode*).

O nome PIM é advindo de sua característica de não ser dependente dos mecanismos fornecidos por algum protocolo de roteamento unicast. Porém, qualquer implementação que suporte o PIM necessita a presença de um protocolo de roteamento unicast

[5]Minimum Spanning Tree

para fornecer informações da tabela de roteamento e para adaptar-se às mudanças na topologia.

Versões dos protocolos PIM foram implementadas para uso com o SIMMCAST, processo este descrito no Capítulo 6. Abaixo segue uma breve descrição dos protocolos, a ser complementada posteriormente.

PIM/DM

O PIM faz uma clara distinção entre um protocolo de roteamento multicast projetado para ambientes densos e outro projetado para ambientes esparsos. O modo denso, descrito em [17], refere-se ao protocolo projetado para operar num ambiente onde os membros do grupo estão densamente distribuídos e a banda passante é abundante. Neste tipo de configuração, faz sentido inundar a rede com pacotes e limitar o tráfego através do esquema de *prune* e *graft*.

O protocolo PIM/DM encaminha todo o tráfego multicast para todas as interfaces de saída do roteador, até que uma mensagem explícita de *prune* seja recebida. Para os casos em que membros do grupo multicast subitamente aparecem em um ramo "podado" da árvore de distribuição, o PIM/DM, assim como o DVMRP, emprega mensagens *grafting* para adicionar o ramo novamente à árvore de distribuição. O resultado final será uma árvore de caminhos da origem a todos os componentes do grupo.

PIM/SM

O PIM modo esparso, descrito em [18, 15], refere-se ao protocolo otimizado para ambientes nos quais os membros do grupo estão distribuídos através de muitas regiões da Internet e a banda passante não é, necessariamente, largamente disponível. É importante notar que o modo esparso não significa que o grupo tem poucos membros, e sim que estes estão muito dispersos pela Internet. A idéia é construir uma árvore enraizada no fonte sem a sobrecarga de gerência típica do DVMRP. É utilizado um ponto central que recebe todos os pacotes endereçados ao grupo e os distribui de acordo com uma árvore multicast tradicional.

Protocolos esparsos, como o PIM, utilizam um roteador intermediário como um "ponto de encontro" entre a origem e o destino. Esse roteador é chamado de *rendezvous point* (RP), ou roteador núcleo. Cada grupo tem um RP e os roteadores tem que descobrir quais são os RPs. Para isso, utilizam um protocolo de *"bootstrap"* ([16]). Os *hosts* que desejam receber os pacotes de determinado grupo mandam mensagens *join* para o RP correspondente. O caminho entre o *host* e o RP é marcado como sendo um caminho para os pacotes. A origem manda os pacotes multicast encapsulados em pacotes unicast para o RP. Se existem *hosts* que mandaram mensagens de *join* para aquele grupo, o encapsulamento é retirado e o pacote é distribuído segundo uma árvore multicast de menor caminho. Se não houver nenhum *host*, o RP manda uma mensagem para a origem, a fim de que ela pare de enviar pacotes, economizando banda.

Apesar da árvore RP ser suficiente para a entrega de pacotes multicast, o aumento do número de participantes pode sobrecarrega-la com um tráfego excessivo. Por

questão de eficiência, PIM/SM não funciona necessariamente com um esquema de árvore compartilhada. Quando um fonte gera tráfego superior a um limite pré-definido (*threshold*), o PIM/SM no DR para de encaminhar pacotes ao RP, adotando, ao invés, uma árvore de menor caminho baseada no fonte (ou seja, nele mesmo). Assim, um fluxo de pacotes que não é negligível passa a trafegar pela rede sem precisar passar pelo RP. Logo, os roteadores que implementam o PIM-SM utilizam a árvore RP, mas também tem a opção de transmitir através de árvores de menores caminhos entre determinado fonte e seus destinatários.

O PIM/SM está sendo desenvolvido para fornecer um protocolo de roteamento multicast que forneça uma comunicação eficiente entre membros de grupos esparsamente distribuídos, que são os grupos mais comuns em grandes redes. Acredita-se que uma situação na qual muitos *hosts* desejam participar em uma conferência multicast não justifica inundar a rede inteira periodicamente com tráfego multicast ([18]). Teme-se que os protocolos de roteamento multicast existentes experimentem problemas de escalabilidade se muitas centenas de pequenas conferências estiverem em andamento, criando grandes quantidades de tráfego agregado que podem, potencialmente, saturar a maioria das conexões Internet. Para eliminar essas potenciais questões de escalabilidade, o PIM/SM foi projetado para limitar o tráfego multicast de modo que apenas os roteadores interessados em receber o tráfego para um grupo em particular sejam capazes de *"vê-lo"*.

O PIM/SM difere dos protocolos multicast modo denso existentes em dois pontos essenciais. Primeiramente, os roteadores que levam a membros do grupo ou que tenham membros diretamente conectados a eles devem se associar à árvore de distribuição de modo esparso pela transmissão explícita de mensagens de associação. Se um roteador não se tornar parte de um árvore de distribuição pré-definida, ele não receberá o tráfego multicast endereçado ao grupo. Já os protocolos de roteamento multicast de modo denso, por sua vez, assumem que há membros do grupo em ramos que saem dele na árvore de distribuição e continua a enviar tráfego multicast por suas interfaces de saída até que mensagens *pruning* explícitas sejam recebidas. Estes modelos foram denominados de *pull* e *push*, respectivamente. A ação padrão de envio dos protocolos de roteamento multicast de modo denso é a de passar adiante o tráfego, enquanto no caso do protocolo de roteamento de modo esparso é a de bloquear o tráfego, a menos que o contrário seja explicitamente solicitado.

2.2.4 Protocolo CBT

O protocolo intra-domínio CBT, especificado em [3, 2], diferentemente do DVMRP ou MOSPF, que possuem uma árvore de caminhos para cada par (origem, grupo destino), constrói uma única árvore compartilhada para cada grupo independentemente do número de fontes no grupo. Um roteador, ou um conjunto de roteadores são escolhidos para formarem o centro de distribuição dos pacotes multicast. O tráfego multicast numa árvore é o mesmo para todo o grupo, não importando quem é a origem. Todas as mensagens destinadas a um determinado grupo são replicadas através do centro de distribuição como se fossem mensagens unicast, até encontrarem um roteador que faça parte da árvore de distribuição correspondente. Os pacotes são, então, replicados, em multicast, em todas as interfaces deste roteador,

exceto aquela por onde foi recebido. O CBT permite que exista mais de um roteador central em cada grupo, fazendo com que haja, na verdade, uma árvore central, que espalha a informação em todas as folhas (membro do grupo).

O CBT foi desenvolvido com o intuito do promover maior escalabilidade, pois como há apenas uma árvore para cada grupo, o uso de memória é diminuído, além de economizar banda passante, já que não há necessidade de mensagens de atualização do grupo periodicamente. Em contrapartida, CBT tem três desvantagens ([5]):

- a latência global é maior, pois para muitos receptores o núcleo não fará parte do menor caminho;
- todos os transmissores usam a mesma árvore de distribuição, criando uma concentração de tráfego ao redor do núcleo;
- o núcleo se constitui em um ponto central de falha.

O PIM/SM é parecido com a abordagem CBT na medida em que também emprega o conceito de *rendezvous point*, onde os receptores se "encontram" com as origens. O iniciador de cada grupo multicast seleciona um RP primário e um pequeno conjunto de RPs alternativos ordenados, conhecido como lista-RP.

Para cada grupo multicast, há apenas um único RP ativo. Cada receptor que desejar se associar a um grupo multicast contata seu roteador diretamente conectado, o qual, por sua vez, se associa à árvore de distribuição ao enviar uma explícita mensagem de associação para o RP primário do grupo. Uma origem utiliza o RP para anunciar sua presença e para encontrar um caminho para os membros que tenham se associado ao grupo.

No entanto, o PIM/SM é mais flexível e eficiente que o CBT, pois enquanto as árvores CBT são compartilhadas pelo grupo, o PIM/SM deixa livre a escolha por uma árvore compartilhada ou de menores distâncias, por parte dos computadores do grupo destino.

2.3 Protocolos Inter-Domínios

Protocolos inter-domínios são executados por roteadores localizados nos limites dos domínios. Uma coleção de domínios, sua política e relacionamentos fim-a-fim e endereçamento definem um sistema de roteamento inter-domínio na Internet.

Um domínio multicast pode abranger uma ou mais árvores de distribuição. Dentro de um domínio, um protocolo de roteamento multicast é usado para manter atualizadas as rotas da(s) árvore(s) de distribuição. Adicionalmente, a idéia de domínio multicast é desassociada da idéia de SA, ou seja, dois ou mais domínios multicast podem estar definidos em SAs diferentes ou não.

O sistema de roteamento inter-domínio representa a base das comunicações na Internet. Duas características influenciam no desempenho destas comunicações ([25]):

- topologia inter-domínio: é o grafo dos domínios e inter-domínios e suas conexões. Um *link* neste grafo significa que existe rota de tráfego entre os correspondentes domínios.

- estabilidade da rota: o sistema de roteamento experimenta alternativas de rotas, ocasionadas por falha em algum roteador ou *link* da rota pré-estabelecida.

Uma análise da topologia do sistema de roteamento e da estabilidade da rota permitem entender como estas características influenciam no desempenho das comunicações. Grandes domínios conectados, causados pelo crescimento da rede, podem aumentar a redundância de conexões fim-a-fim. Mesmo assim, o crescimento da Internet pode aumentar o atraso de propagação e diminuir drasticamente o *throughput* de aplicações distribuídas, o que pode levar a falhas de temporização ou suspeitas incorretas de colapsos de processos. O aumento da perda de pacotes ou o re-ordenamento destes podem reduzir a eficiência da rede como um todo. Diversas características precisam ser atendidas na implementação de protocolos de roteamento inter-domínios. Independente da topologia existente, protocolos de comunicação multicast inter-domínios precisam ser escaláveis em toda a rede. Este e outros requisitos para roteamentos inter-domínios, analisados em ([35]), sob o aspecto de um sistema distribuído, são apresentados a seguir:

- escalabilidade: a habilidade de um protocolo escalar é crucial em roteamentos inter-domínios. A escalabilidade de um protocolo determina se ele pode reunir um grande número de receptores ou está limitado à determinado número de usuários simultâneos. O uso de recursos como memória nos roteadores, assim como o tamanho da tabela de roteamento e uso da largura de banda, são algumas medidas que permitem avaliar a escalabilidade de um protocolo.

- estabilidade: o algoritmo de roteamento tem de convergir rapidamente, sendo convergir equivalente a atingir um estado correto. Por exemplo, quando acontece alguma modificação na topologia da rede, as tabelas de roteamento de alguns roteadores apresentam informações desatualizadas, temporariamente inconsistentes em relação ao estado físico da rede. O algoritmo "converge" quando todos os roteadores tiverem atualizado consistentemente suas tabelas de rotas em função de um estímulo externo inicial, tal como a queda de um *link*. No entanto, árvores de distribuição não devem ser atualizadas frequentemente, uma vez que isto causa um aumento no tráfego de mensagens na rede, bem como uma potencial perda de pacotes nas transações em progresso ([64]). Segundo [35], acredita-se que reduzir a sobrecarga do protocolo é melhor do que manter a árvore de distribuição sempre ótima.

- robustez: uma vez que a rede entre em operação, deve permanecer assim durante anos, sem que ocorram falhas de todo o sistema. Durante este período, ocorrerão falhas isoladas de hardware e software e a topologia da rede modificar-se-á diversas vezes. O algoritmo de roteamento deve ser capaz de resolver estas modificações sem necessitar uma reinicialização.

- latência na entrega: latência representa o tempo que um pacote demora para ser entregue a seus receptores. A latência deve ser mínima em aplicações que

envolvam comunicações como áudio e vídeo conferência. No entanto, aplicações como *download* de arquivos da Internet são tolerantes a tempos de latência maiores nas transmissões.

- restrições de roteamento: é importante para um protocolo de roteamento inter-domínio possuir um modelo de policiamento que controle o fluxo do tráfego multicast que trafega na Internet. O suporte à restrições de roteamento envolve dois aspectos: (1) política de restrições, que é específica de cada domínio e (2) restrições de perda, latência, *jitter* ou outra métrica dinâmica. Isto é utilizado em roteamentos com Qualidade de Serviço (QoS).

- balanceamento de carga: a falta de um mecanismo de balanceamento de carga pode levar a situações em que um *link* inter-domínio esteja muito saturado com tráfego multicast, enquanto outros *links* estão sem uso.

- roteamento independente: a independência do protocolo de roteamento multicast inter-domínio permite que cada domínio escolha o protocolo intra-domínio para executar dentro do domínio. Isto garante autonomia para cada domínio selecionar o protocolo que melhor atenda às suas necessidades. Isto também permite que um domínio faça atualizações de versão do protocolo de roteamento minimizando os efeitos para outros domínios.

Para que os membros de um grupo multicast possam receber tráfego de fontes em outros domínios, é necessário um protocolo que troque informações de fontes ativas e seus grupos multicast correspondentes em cada árvore.

Os principais protocolos de roteamento multicast inter-domínios na Internet, que permitem que pacotes sejam encaminhados do fonte aos múltiplos destinos através da rede são:

- MSDP (*Multicast Source Distribuition Protocol*);
- BGMP (*Border Gateway Multicast Protocol*);
- MBGP (*Multiprotocol Extensions for BGP-4*).

2.3.1 Protocolo MSDP

MSDP, descrito em [66], é um protocolo inter-domínio que conecta múltiplos domínios PIM/SM. O protocolo MSDP permite que diversos domínios PIM/SM compartilhem informações sobre os fontes ativos na rede. Cada domínio PIM/SM usa seus próprios *rendezvous points* (RP) independentes de RPs em outros domínios.

A comunicação é implementada via uma conexão TCP. Através dela, informações de controle são trocadas entre os pares MSDP. Se um RP de um domínio PIM/SM recebe uma unidade de registro PIM, ele irá registrar este fonte e enviar esta informação para seus pares.

Quando a troca de informações MSDP acontece entre domínios definidos em diferentes SA, então, além da troca de fontes e grupos ativos efetuados pelo MSDP,

há a necessidade dos roteadores trocarem informações específicas sobre roteamento multicast. Para isto, utilizam o protocolo BGMP, que fornece um método para os roteadores determinarem os caminhos que as árvores de distribuição multicast irão seguir para a entrega dos pacotes multicast, da fonte para o(s) receptor(es) aprendido(s) pelo MSDP. A Figura 2.4 representa a operação do protocolo MSDP.

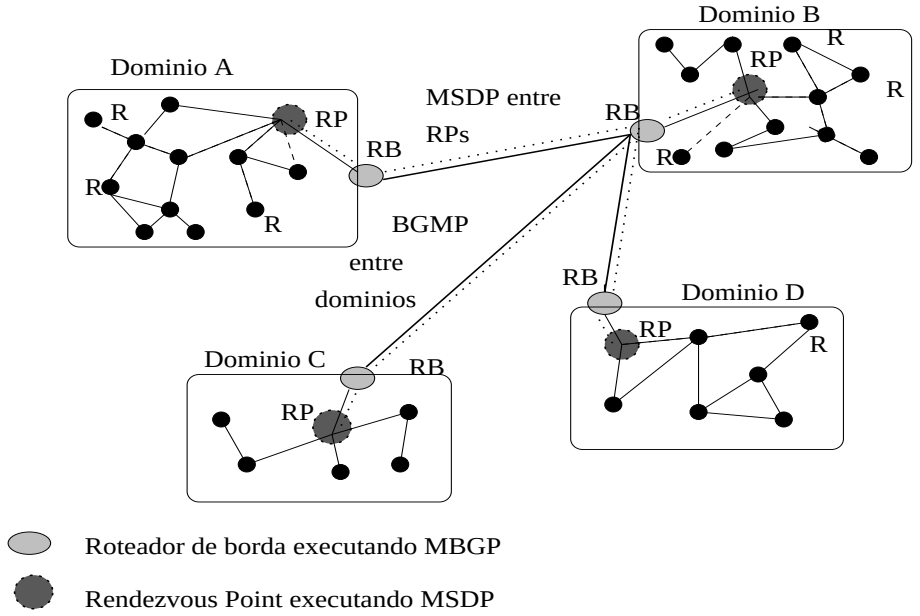


FIGURA 2.4 – Operação do Protocolo MSDP

Quando fontes multicast começam a transmitir, a rede deve criar algum tipo de controle de estado do roteamento, visando o controle do fluxo de pacotes. Informações sobre a existência de fontes ativos precisam ter sido criadas antes do registro do estado do roteamento. Esta complexidade extra aumenta a sobrecarga de gerência de grupos.

O protocolo MSDP ainda apresenta problemas de escalabilidade, pois todos os RPs existentes na rede precisam "escutar" todos os fontes conectados. A limitação ocorre se o uso de multicast cresce para um número da ordem de milhares de fontes multicast. Como mensagens de informação de fontes ativos são enviadas periodicamente, o número de mensagens (dado extra) inundando a rede tornar-se-á excessivo.

2.3.2 Protocolo BGMP

É um protocolo baseado na escolha de um domínio raiz ou núcleo, e em uma árvore compartilhada envolvendo os roteadores de borda. O BGMP permite a interação entre domínios com diferentes protocolos ([35, 59]).

A principal característica do protocolo BGMP é construir uma árvore bidirecional compartilhada entre domínios, utilizando um domínio raiz. O uso de um domínio raiz implica em possível ponto de falha no sistema. Ainda, pode-se afirmar que há uma concentração do número de mensagens trafegando junto ao domínio raiz.

BGMP foi desenvolvido utilizando conceitos dos protocolos CBT e PIM/SM, pois estes dois protocolos utilizam RP. No protocolo BGMP a seleção do domínio raiz é feita com base no endereço multicast.

BGMP assume que diferentes faixas de endereços IP multicast estão associados à diferentes domínios. Se um receptor quer se unir ao grupo, o roteador de borda do domínio deste receptor gera uma mensagem específica de *join* para aquele grupo, a qual é encaminhada através do roteador de borda até que ela alcance, ou o domínio raiz ou um ramo da árvore BGMP. Todos os roteadores da rota criam um grupo específico, bidirecional, de modo que qualquer pacote multicast destinado àquele grupo é encaminhado na árvore BGMP ([48]). Uma importante característica do protocolo BGMP é permitir que ramos uni-direcionais possam ser conectados à árvore bi-direcional, e que, através do BGMP, herdem características do PIM e do CBT.

A Figura 2.5 apresenta uma árvore compartilhada de domínios em ação. Neste exemplo, o Domínio B quer alocar um grupo multicast que inclua o endereço 224.2.2.2 e é, então, designado como o raiz deste grupo. Ainda, os domínios E, F e G precisam se unir a este grupo. Este processo resulta no envio de mensagens BGMP *join* pelos roteadores de borda destes domínios para o domínio raiz. A Figura 2.6 apresenta a árvore compartilhada de domínios resultante dos *joins* BGMP enviados ([65]).

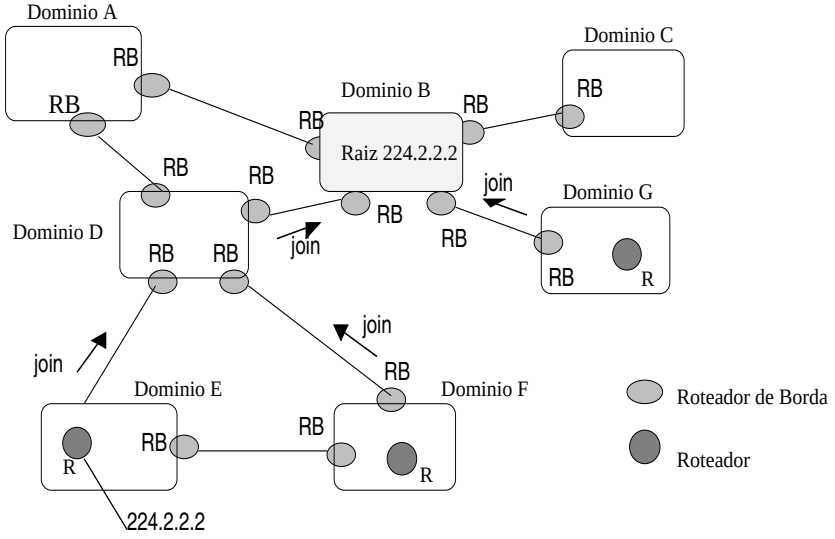


FIGURA 2.5 – Árvore BGMP com Domínios em Ação

2.3.3 Protocolo MBGP

O protocolo inter-domínio MBGP ([7]) é uma extensão do protocolo de envio intra-domínio BGP-4 ([52]). O protocolo MBGP adicionou novos recursos a fim de permitir multicast intra e inter SA na Internet. MBGP permite aos roteadores determinarem os caminhos que as árvores de distribuição multicast irão seguir para a entrega dos pacotes multicast.

Com MBGP, ao invés de cada roteador precisar conhecer a topologia da rede multicast, cada roteador apenas precisa conhecer a topologia de seu próprio domínio e os

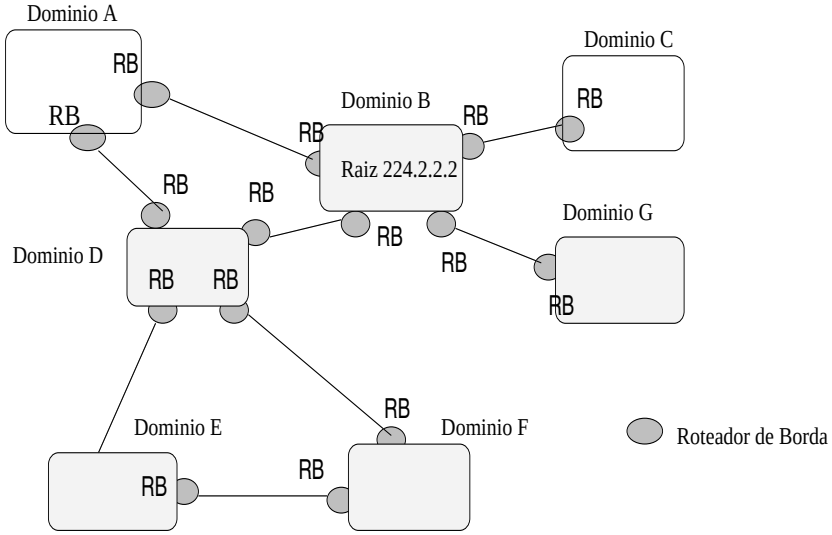


FIGURA 2.6 – Árvore BGMP Resultante

caminhos para procurar os outros domínios ([1]). A Figura 2.7 mostra um exemplo de topologia multicast inter-domínio executando BGP e/ou MBGP.

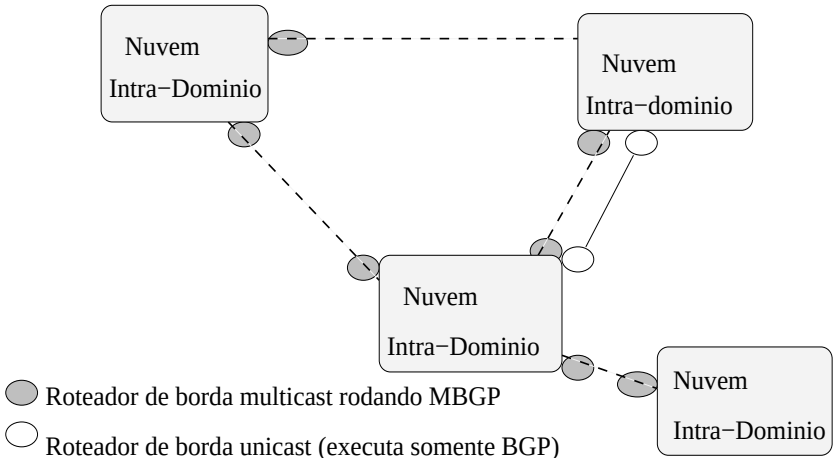


FIGURA 2.7 – Domínios Conectados Através de Sessões MBGP

O MBGP transporta dois conjuntos de rotas: um conjunto para o roteamento unicast e outro para o roteamento de pacotes multicast. As rotas associadas no roteador são mantidas atualizadas, a fim de permitir ao roteador tomar decisões de encaminhamento dos pacotes multicast nas bordas inter-domínios.

Duas características importantes são associadas ao protocolo MBGP:

- causa um aumento no tamanho das tabelas de roteamento;
- possui um problema potencial de armazenamento redundante de informações, pois mútiplos conjuntos de rotas para os mesmos prefixos podem ser armazenados no roteador.

O protocolo MBGP também apresenta problemas de escalabilidade, se analisado o aspecto referente ao consumo de recursos de memória no roteador.

2.4 Desempenho de Algoritmos de Roteamento

Uma comparação quantitativa de algoritmos de roteamento é bastante complexa por uma série de razões. Uma delas está associada aos diversos critérios possíveis de análise de desempenho existentes. Muitas vezes um algoritmo tem um melhor desempenho para determinados objetivos, enquanto outros podem ser melhores se trocarmos os objetivos. Outra razão se deve ao fato de que a desempenho do algoritmo depende da configuração específica da rede.

Quando se está comparando desempenho de algoritmos de mesmo tipo (por exemplo, algoritmos de caminho mais curto), o que mais interessa são as características do mesmo associadas à sua capacidade de se adaptar à mudanças na rede. Alguns critérios que podem ser utilizados são analisados a seguir ([56]):

- tempo de convergência: define-se tempo de convergência como sendo o tempo necessário para que uma mudança na topologia se propague através da rede e para que as novas tabelas de roteamento estabeleçam um novo caminho mais curto. Uma possível medida de velocidade é o número de iterações necessárias para que o algoritmo convirja para um novo estado. Esta medida de desempenho é extremamente importante em um ambiente dinâmico. O tempo de convergência deve ser menor que a taxa de mudança na rede, caso contrário, a convergência não ocorrerá e o algoritmo de roteamento será inútil.

- número de pacotes de controle transmitidos: tem-se pacote de controle como a informação que deve ser transmitida ao longo da rede para propagar as modificações ocorridas, as quais podem ser modificações topológicas ou nos custos associados, e as atualizações das tabelas de roteamento. Quanto maior o número de pacotes de controle, maior a sobrecarga introduzida e maior a possibilidade de congestionamento gerada pelos pacotes de controle.

- complexidade computacional: os algoritmos necessitam um esforço computacional variável para processar os pacotes de controle, executar o algoritmo propriamente dito e atualizar as tabelas de roteamento. Isto pode refletir no tempo de execução para cada nodo, e portanto, na velocidade de processamento do protocolo de roteamento.

- tamanho dos pacotes de controle: os algoritmos diferem na quantidade de informação transmitida por pacote de controle.

- espaço de buffer: os algorimos diferem no espaço de memória necessário para atualizar e manter as tabelas de roteamento. Esta propriedade está diretamente relacionada com a escalabidade do protocolo de roteamento.

- *loop:* no estado transitório ou de convergência de um algoritmo, pode-se ter a ocorrência de *loops*. Um algoritmo *loop free* garante que os pacotes não retornarão aos nodos pelos quais eles já passaram, mesmo quando as tabelas de roteamento estão em processo de mudança.

Como já mencionado, duas medidas de desempenho são substancialmente afetadas pelos algorimos de roteamento: o *thoughput*, que representa a quantidade de serviço,

e o atraso médio sofrido pelo pacote, que indica a qualidade do serviço. O roteamento interage com o controle de fluxo na determinação destas medidas por meio de um mecanismo de realimentação. Quando o tráfego chegando à subrede (tráfego oferecido) é relativamente baixo, ele é completamente aceito pela rede, isto é, $throughput = trafego\ oferecido$. Quando o tráfego oferecido é excessivo, uma parte dele será rejeitado pelo controle de fluxo, e $throughput = trafego\ oferecido - trafego\ rejeitado$.

O tráfego aceito na rede sofrerá um atraso médio por pacote, o qual dependerá da rota escolhida pelo algoritmo de roteamento. Contudo, o *thoughput* também será afetado, mesmo que indiretamente, pelo algoritmo de roteamento, uma vez que os esquemas de controle de fluxo típicos operam na base de um balanceamento entre *thoughput* e atraso, isto é, eles iniciam a rejeição do tráfego oferecido quando o atraso torna-se excessivo.

Desta forma, quanto mais sucesso o algoritmo de roteamento tiver em manter o atraso baixo, mais tráfego de entrada o algoritmo de controle de fluxo permitirá. Embora o balanceamento preciso entre atraso e *throughput* seja determinado pelo controle de fluxo, o efeito de um bom roteamento, sob condições de tráfego oferecido alto, oferece uma relação *thoughput* x atraso mais favorável, sob a qual o controle de fluxo opera.

2.5 Comentários Adicionais

Este capítulo apresentou uma revisão dos principais protocolos de roteamento multicast em uso na Internet. Definiu conceitos fundamentais para o cerne do trabalho, como roteamento, tipos de domínio, intra e inter-domínios, além de questões envolvendo análise de desempenho e o modo de execução dos referidos protocolos.

Capítulo 3

SIMULADORES DE REDES

O uso de simuladores de rede é de grande valia no processo de projetar e avaliar protocolos de comunicação. Este capítulo traz uma revisão bibliográfica relativa a ambientes de simulação de redes de computadores. Inicialmente, a Seção 3.1 define simulação e apresenta conceitos relacionados a simuladores de rede, enquanto que as Seções 3.2 e 3.3 apresentam exemplos representativos deste tipo de ferramenta para o cerne do trabalho.

3.1 Simulação

Simulação é a imitação da operação de um processo do mundo real, sendo utilizada para descrever e analisar o comportamento de um sistema ([4]).

O uso de simulação tem sido uma ferramenta poderosa no processo de projetar e avaliar protocolos de comunicação. Segundo Jain ([30]), simulação é uma técnica para análise de desempenho de sistemas computacionais. O estudo via simulação permite comparar resultados obtidos em uma variedade de ambientes, de modo a ajudar em processos de tomada de decisão. Selecionar a linguagem de programação é uma etapa importante no processo de modelagem da simulação. Uma decisão errada nesta etapa do processo pode levar ao aumento do tempo de desenvolvimento da simulação, estudos incompletos e falhas na obtenção dos resultados.

Além de simulação, existem outros dois meios de se obter conhecimento sobre um protocolo: avaliação analítica e experimentos. Estas três formas são complementares e podem representar diferentes partes do mesmo processo de desenvolvimento ([9]). A avaliação analítica é muito útil para demonstrar o impacto geral dos argumentos de entrada de um protocolo (como o tamanho da janela), do número de participantes, ou de mudanças em condições específicas da rede (como a taxa de perda). Entretanto, ela necessita modelos extremamente simplificados da realidade, limitando o número de variáveis consideradas e as fontes de não-determinismo, já que os resultados são obtidos alterando argumentos em fórmulas. Avaliação analítica é usada principalmente para determinar tendências gerais no comportamento de modelos abstratos de protocolos, ou para provar formalmente a correção de um protocolo.

Em contraste, experimentos práticos são baseados em um conjunto de execuções de

um protocolo sobre uma rede real, e portanto, produzem resultados mais precisos. Apesar disso, devido à grande influência da topologia e dos sistemas e configurações envolvidos, os resultados coletados são dependentes de um cenário específico, e são usualmente difíceis de serem reproduzidos em outro. Isto se aplica particularmente a protocolos multicast escaláveis, já que é inadequado realizar experimentos de larga escala em uma rede pública como a Internet. Outra limitação considerável é que esta abordagem implica que o protocolo já esteja implementado antes que seu funcionamento geral possa ser testado.

A simulação se encontra no nível intermediário, entre a avaliação analítica e os experimentos práticos. Ela permite que o projetista do protocolo ajuste o nível de detalhe, atendo-se apenas aos recursos desejados. Além disso, simulação possui o potencial de permitir o aumento gradual de detalhamento, para que o processo resulte em um protocolo executando sobre uma rede emulada, pronto para ser movido para uma rede real.

O modelo a ser simulado representa uma visão abstrata do protocolo e da rede sob este (com sua topologia, conectividade, correlação de perda de pacote, padrões de tráfego, etc). Construir um modelo de simulação envolve partir de premissas que simplifiquem o cenário de modo a focalizar-se nos aspectos relevantes do estudo ([26]). A precisão dos resultados obtidos dependerá de quão válidas as premissas iniciais eram. Desta forma, o primeiro passo na construção de um modelo de simulação de um sistema é determinar exatamente quais são as características importantes e que precisam ser mensuradas, bem como quais as variáveis do sistema podem ser afetadas por elas.

Apesar de ser uma técnica bastante poderosa para a avaliação de sistemas, um modelo de simulação pode produzir resultados não confiáveis ou errados. Na sequência são apresentados os erros mais comuns cometidos em simulações ([30]):

- nível de detalhes inapropriado: a simulação permite o estudo de um sistema de forma mais detalhada que a modelagem analítica, que necessita geralmente várias considerações para tornar o problema tratável matematicamente. Em uma simulação o nível de detalhes é limitado pelo tempo de desenvolvimento disponível. Além disso, quanto mais detalhes, maior a probabilidade de ocorrerem erros e mais difícil se torna encontrá-los. Outro aspecto a ser considerado é o tempo de execução necessário e o hardware disponível. Quanto maior o detalhamento, maior a quantidade de parâmetros de entrada e muitas vezes os valores desses parâmetros não são conhecidos. Desta forma, um maior nível de detalhes não produz necessariamente uma simulação melhor.

- linguagem de programação inadequada: a escolha da linguagem de programação tem um impacto considerável no tempo de desenvolvimento do modelo. Linguagens de propósito geral são portáveis e proporcionam maior controle sobre a eficiência e o tempo de execução da simulação. Por outro lado, linguagens específicas para simulação necessitam menor tempo de desenvolvimento e incluem ferramentas para tarefas comuns como análise estatística, por exemplo.

- modelos incorretos: os modelos de simulações são geralmente programas extensos, nos quais erros de programação são bastante comuns, o que pode resultar em resultados comprometidos. Mesmo que o programa não contenha erro, este pode não representar o sistema de forma adequada, devido a considerações incorretas sobre o comportamento do sistema.

- inexistência ou má definição de objetivos: uma simulação é um projeto complexo e exige uma especificação clara dos objetivos. Os objetivos devem ser escritos e estarem claros para todos os envolvidos no processo de desenvolvimento.

- tratamento incorreto das condições iniciais, finais e tempo de simulação: na maioria dos sistemas reais existe um estado transitório antes do sistema atingir o equilíbrio. Nas simulações também existe um estado inicial que não representa o comportamento do sistema em estado estacionário. Esta parte inicial da simulação não deve ser considerada.

- geradores de números aleatórios: modelos de simulação utilizam variáveis aleatórias para representar o comportamento de um determinado parâmetro do sistema, como por exemplo, o tamanho dos pacotes gerados por uma determinada aplicação. Esses valores aleatórios são gerados a partir de procedimentos chamados geradores de números aleatórios. É mais seguro utilizar um gerador bastante conhecido e analisado do que implementar o próprio gerador.

- escolha inadequada de sementes: os geradores de números aleatórios geram cada valor a partir de um valor anterior. Desta forma, para gerar uma seqüência de números aleatórios o programa deve receber como entrada um valor inicial, chamado de semente. As sementes para diferentes seqüências aleatórias devem ser escolhidas cuidadosamente, de modo que seja mantida a independência entre as seqüências. Utilizar a mesma semente ou compartilhar uma seqüência de números aleatórios entre diferentes processos são erros bastante comuns que levam a conclusões que podem representar de forma incorreta o comportamento do sistema.

Um simulador de protocolos, tipicamente, oferece um conjunto de interfaces para a configuração de uma simulação e para a escolha do tipo de escalonador de eventos utilizado para conduzir a simulação.

O arquivo de definição de uma simulação, normalmente, inicia criando uma instância desta classe e chamando métodos para a criação de nodos, topologias e para a configuração de outros aspectos da simulação. As topologias são formadas através da criação dos nodos e da conexão destes por enlaces.

Os seguintes tipos de escalonadores de eventos podem estar disponíveis: (1) orientado a eventos, onde há um procedimento associado a cada tipo de evento no sistema; e (2) orientado a processos, onde o caminho de operação é obtido pela interação do número dos processos que executam em paralelo. A função do escalonador é selecionar o próximo evento, executá-lo até o fim e retornar para executar o próximo evento ([37]). O escalonador executa apenas um evento por vez. Se mais de um evento for escalonado para ser executado, estes serão ordenados e executados em série.

Simuladores de redes permitem avaliar protocolos de comunicação sob diferentes condições da rede, sob as quais pesquisadores podem submeter seus protocolos. São elas ([4]):

- abstração: variar a granularidade da simulação permite avaliar um protocolo sob diferentes ênfases, desde aspectos macroscópicos da análise, até a avaliação detalhada do protocolo;
- emulação: permite a execução de um simulador para iteragir com os nós da rede real;
- geração de cenários: permite o teste de protocolos sob diferentes condições de rede, necessário para uma completa avaliação do protocolo ([10]). A criação automática de diferentes condições de tráfego, topologias e eventos dinâmicos como a falha em um *link* de comunicação ajudam a criar diferentes cenários de simulação;
- visualização: permite a visualização gráfica dos resultados obtidos durante a simulação, com recursos que podem incluir o uso de cores, rótulos e interatividade;
- modularidade: facilita a inclusão de novas funcionalidades na simulação.

Um estudo baseado em simulação não pode ser considerado como um processo sequencial simples, principalmente porque um estudo deste tipo, além dos resultados de simulação, oferece a oportunidade de entender mais profundamente o sistema em avaliação.

O rápido crescimento da Internet, tanto em número de nós como na variedade de serviços, tem motivado o desenvolvimento de novos protocolos e algoritmos para que se tenham atendidos os requisitos necessários em comunicações em rede. Existe uma gama considerável de trabalhos em simulação de redes de computadores, resultando em diversas ferramentas. Estas podem ser de forma geral divididas em, no que se convencionou neste trabalho denominar, simuladores e ambientes de teste[1]. Ambientes de teste permitem que certas condições sejam emuladas filtrando, atrasando ou duplicando pacotes em um *host* da rede. Alguns exemplos destes ambientes são o Delayline([29]) e o Dummynet ([54]).

Simuladores permitem que qualquer condição específica conhecida seja testada; em geral, o desenvolvedor tem muito mais controle sobre o experimento com simuladores do que com ambientes de teste. No entanto, recursos finitos de computação, como memória e tempo de processamento, limitam o número de objetos de rede, ou seja, as dimensões dos experimentos que os projetistas podem simular. Na sequência são descritos os simuladores NS, tido como referência entre eles, e JAVASIM, biblioteca base do desenvolvimento do simulador SIMMCAST.

[1] Testbeds

3.2 VINT ns

O simulador ns ([20]) é um simulador de rede em desenvolvimento no projeto *Virtual InterNet Testbed* (VINT), uma colaboração entre UC Berkeley, LBL, USC/ISI e Xerox PARC.

O simulador ns tem sido usado principalmente na pesquisa de desempenho do TCP, mas também em pesquisas de dinâmica quanto à interação de múltiplos protocolos, políticas de enfileiramento em roteadores, protocolos multimídia, protocolos de multicast confiável e controle de congestionamento.

O ns é um simulador orientado a objetos escrito em C++ e OTcl. Os usuários geram scripts na linguagem OTcl (um dialeto de Tcl, vide [47]) que são interpretados e executados pelo simulador. O simulador suporta uma hierarquia de classes em C++ e uma hierarquia de classes similar para o interpretador OTcl.

O simulador, descrito por uma classe em Tcl `class Simulator`, oferece um conjunto de interfaces para a configuração de uma simulacão e para a escolha do tipo de escalonador de eventos utilizado para conduzir a simulação. Um script de simulação, normalmente, inicia criando uma instância desta classe e chamando métodos para a criação de nodos, topologias e para a configuração de outros aspectos da simulação.

O ns possui bibliotecas com uma quantidade considerável de protocolos implementados e funções de controle da simulação para escalonamento de eventos. O núcleo do simulador implementa tarefas que exigem processamento de alto desempenho como o tratamento de eventos de baixo nível e encaminhamento de pacotes através de um roteador simulado.

Para o desenvolvimento de pesquisas com o ns, os usuários criam novos objetos do simulador através do interpretador que instancia estes objetos e gera um espelho em C++, criando um objeto correspondente. Desta forma, o usuário deve estar a par dos componentes usados em cada fase do processo de simulação, desde a concepção do modelo até a análise dos resultados. Inicialmente, o usuário deve implementar seu modelo de simulação utilizando Otcl a partir dos objetos (ex: protocolos, aplicações) existentes na biblioteca ns. Quando o usuário desejar implementar novos protocolos ou estender os já existentes na biblioteca do ns, deverá utilizar uma combinação de C++ OTcl, e, posteriormente, usar as novas funções através de chamadas em scripts OTcl. O programa de simulação desenvolvido alimentará um interpretador OTcl e serão gerados arquivos de traço contendo os resultados da simulação. Comandos e métricas a serem coletadas para a geração da saída da simulação deverão ser explicitamente incluídos no script OTcl.

Apesar de agregar diversas tecnologias, o ns apresenta algumas limitações. O fato de ter sido originalmente desenvolvido para o estudo da Internet, especificamente, de redes IP, limita sua aplicação para o estudo de protocolos/tecnologias sub-IP. Outra limitação é a curva de aprendizado difícil e a falta de documentação adequada para iniciantes. O desenvolvimento de modelos simples implica em conhecimento de Otcl que não é complexa, mas pouco conhecida. À medida que se necessita desenvolver novos protocolos ou mecanismos não presentes no ns, deve-se ter não

apenas intimidade com o desenvolvimento em C++, mas desvendar a estrutura de dados do ns e seu esquema de hierarquias de classes para saber de onde novas classes podem ser derivadas e encaixadas, o que não é uma tarefa simples para um iniciante no ns.

Diversos tipos de escalonadores de eventos estão disponíveis no ns, cada um tendo sido implementado utilizando uma estrutura de dados diferente, como, por exemplo, lista simplesmente encadeada. A função do escalonador é selecionar o próximo evento, executá-lo até o fim e retornar para executar o próximo evento. O escalonador do ns executa apenas um evento por vez, e, se mais de uma evento forem escalonados para serem executados ao mesmo tempo, estes eventos são ordenados e executados em série.

As topologias são criadas no ns através da definição de nodos e da conexão destes por enlaces. A função de um nodo, quando recebe um pacote, é examinar seus campos, normalmente o endereço destino, mapeando os valores em um objeto de interface de saída, que será o próximo receptor deste pacote.

O ns possui suporte a uma grande quantidade de tecnologias e protocolos reais. Entre eles, disponibiliza um software visualizador de resultados da simulação, o animador nam ([19]). O nam (*Network Animator*) é uma ferramenta importante para a visualização da dinâmica da simulação e para a depuração. Através do nam pode-se visualizar a topologia da rede, bem como acompanhar o fluxo dos pacotes e seus conteúdos. Quando o nam é ativado, apresenta uma console que pode gerenciar várias atividades paralelamente, como animações e criação de simulações. Quando essa console é finalizada, todas as outras janelas são fechadas também. Durante a simulação, o simulador gera um ou mais arquivos de traço que contêm dados detalhados da simulação, para visualização posterior. A criação destes arquivos é opcional nos simuladores. No final da simulação, o nam pode ser acionado para interpretar o arquivo de traço e mostrar a animação da simulação. Além de servir para a animação de modelos de simulação, o nam aceita arquivos formatados de traços de dados gerados por redes reais. Esta é uma característica importante dada a necessidade de obter resultados de desempenho baseados em tráfegos que modelem de forma mais fiel as aplicações existentes.

No entanto, apesar de contar com tantos recursos, o ns possui o código desnecessariamente complexo. Para desenvolver um novo protocolo, o desenvolvedor precisa compreender o funcionamento interno do ns, modificá-lo, e recompilar o ns com o suporte ao novo protocolo. Por exemplo, para criar novos tipos de pacote, arquivos do tipo cabeçalho do ns precisam ser alterados; e, claramente, esta não é uma boa abordagem de API.

Além disso, o ns não suporta a abordagem baseada em processos: logo, ele não tem suporte para protocolos *multi-thread*.

3.3 JavaSim

JavaSim, descrito em [38], é uma biblioteca de classes para simulações de propósito geral. Apesar de não ser um simulador de redes, está aqui descrito devido a ser a biblioteca base do desenvolvimento do simulador Simmcast.

JavaSim é a implementação em Java do *toolkit* de simulação C++SIM (derivado de Simula ([8])), que foi desenvolvido em conseqüência do projeto *Arjuna* ([49]). Desta forma, ambos compartilham os mesmos requisitos, que são:

- ser de fácil aprendizado e uso: a interface da biblioteca do simulador deve ser simples;
- portabilidade facilitada: programadores de outros ambientes de simulação devem julgar a transição para JavaSim como de fácil execução;
- ser flexível e extensível: deve ser fácil para programadores incluir novas funcionalidades ao sistema, tais como novas funções de distribuição;
- eficiência: o sistema deve ser eficiente e produzir simulações que executem de forma eficiente.

A fim de atender os requisitos de projeto, JavaSim foi desenvolvido como um simulador de eventos discretos baseado em processos, que herdou as características providas por Simula, como já mencionado. Isto possibilitou ao JavaSim ser flexível e extensível, permitindo que novas funcionalidades sejam incluídas sem afetar a estrutura do sistema como um todo. Por exemplo, a geração de números randômicos e funções de distribuição de probabilidades podem ser definidas através de especialização de classes já existentes.

Por ser um simulador orientado a processos, cada entidade da simulação é considerada um processo separado, onde entidade representa um objeto que necessita definição explicita ([4]). A cada processo é atribuido um objeto Java que possui uma *thread* de controle independente. Seguindo o paradigma da programação orientada a objetos, e para tornar o desenvolvimento de processos simplificados, é definida uma classe básica que implementa a funcionalidade de processos.

A classe básica é chamada `SimulationProcess` e define todas as operações necessárias para o sistema de simulação controlar as entidades da simulação. Como o construtor desta classe é protegido, não é possível a instanciação desta classe, de modo que os processos devem ser derivados dela. O simulador Simmcast é um exemplo de aplicação que estende esta classe. Desta forma, as características do JavaSim também são absorvidas por ele.

Capítulo 4

O SIMULADOR SIMMCAST

SIMMCAST é um *framework* de simulação de eventos discretos baseado em processos que auxilia no projeto e avaliação de protocolos e sistemas distribuídos, particularmente baseados em multicast [6]. Ele é utilizado em modelos de simulação de eventos discretos, onde sistema de simulação discreto é aquele cujo estado das variáveis muda de forma instantânea em períodos separados de tempo ([37]).

Este capítulo descreve o SIMMCAST, reunindo de forma consistente os artigos existentes sobre o simulador e mostrando sua estrutura de classes através de diagramas UML. A Seção 4.1 traz informações relativas a desenvolvimento de *frameworks*. Já as Seções 4.2 e 4.3 apresentam a arquitetura inicial e os diagramas de classes do simulador. Adicionalmente, a Seção 4.4 traz um breve tutorial sobre como desenvolver simulações com o SIMMCAST.

4.1 Desenvolvimento de *Frameworks*

Protocolos distribuídos implementam algoritmos distribuídos utilizados para realizar uma comunicação e/ou coordenação envolvendo vários *hosts*.

Técnicas de orientação a objetos (OO) tem sido recentemente aplicadas com sucesso ao projeto e à implementação de softwares de comunicação e no desenvolvimento de sistemas distribuídos. Diferentes abordagens tem sido utilizadas durante os últimos anos, correspondendo à diferentes visões ou enfatizando diferentes aspectos da utilização da tecnologia OO. Todas essas visões são baseadas na idéia genérica de que técnicas de implementação e projeto orientado a objetos podem ser utilizadas para aumentar a modularidade e extensibilidade através da definição de interfaces estáveis, que encapsulam os detalhes da implementação ([57]).

Aliados à tecnologia de OO, padrões de projetos[1] ([22]) representam soluções documentadas para problemas de desenvolvimento dentro de um contexto particular. Padrões de projetos capturam experiências consagradas de desenvolvimento e ajudam a promover uma boa prática de projeto.

[1] *design patterns*

Frameworks são aplicações reutilizáveis "semi-completas" que podem ser especializadas para produzir aplicações particulares. Um *framework* aumenta a reusabilidade de software, o que traz vantagens como: redução no esforço de desenvolvimento e um código mais robusto através de múltiplos reusos e refinamentos do *framework* ([31]). Um *framework* pode ser definido sob dois aspectos: estrutura e função. Quanto à estrutura, um *framework* é a reutilização do *design* de parte ou totalidade de um sistema representado por classes abstratas e pela forma como instâncias destas classes interagem. Em outras palavras, a estrutura de um *framework* é mostrada pelo seu diagrama de classes.

Quanto à função, um *framework* é o esqueleto de uma aplicação que pode ser personalizado por um desenvolvedor de aplicação. Ou seja, a função do *framework* é extrair e disponibilizar a essência de uma determinada família de aplicações, de modo que o desenvolvedor possa partir de um patamar superior de implementação da arquitetura. Neste sentido um *framework* tem a mesma função de uma API.

Por outro lado, do ponto de vista de desenvolvimento, um *framework* pode ser visto como uma arquitetura de software reutilizável em termos de contratos de colaboração entre classes abstratas, ou *kernel*, e um conjunto de pontos de flexibilização, ou *hot spots* ([22]).

O *kernel*, ou núcleo do *framework*, fornece elementos básicos de comunicação a nível de pacote e gerenciamento de grupo e não permite alteração pelo usuário. Por outro lado, *hot spots* consistem de pontos que precisam ser especializados em uma aplicação. Os contratos de colaboração definem as regras que tal especialização deve obedecer. Um *hot spot* é implementado a partir de padrões de projeto, e estes consistem na principal documentação de um *framework*.

4.2 O *Framework* SIMMCAST

Frameworks são particularmente apropriados para simulações, uma vez que uma parte substancial do código de uma simulação pode ser aproveitada em outros experimentos.

SIMMCAST é um *framework* de simulação ([6, 44]) que permite que protocolos sejam facilmente definidos por uma combinação de blocos básicos. Ele é orientado a objetos e suporta a especificação de protocolos multicast com múltiplas *threads*. É disponibilizado como um *framework* porque a arquitetura desenvolvida não implementa um tipo particular de protocolo/aplicação, e sim um modelo que é especializado de acordo com a necessidade do usuário.

SIMMCAST permite a especificação de um novo protocolo através da combinação de blocos básicos disponíveis, inserindo neles o código para a lógica do protocolo. Desta forma, modelos abstratos podem ser expressos com o SIMMCAST. A partir daí, podem ser complementados para representar diferentes níveis de detalhe, oferecendo informações preciosas sobre o desempenho real de protocolos multicast.

SIMMCAST é uma camada de software desenvolvida ao redor do mecanismo do JAVA-SIM ([38]), provendo primitivas básicas de comunicação em nível de pacotes e gerenciamento de grupo. Para realizar uma simulação, o usuário deve adicionar/estender classes do *framework* de acordo com o protocolo e a configuração que deseja avaliar.

O propósito do SIMMCAST é prover um *framework* com operações específicas de comunicação e temporização, bem como o suporte para projeto e avaliação de protocolos multicast. Para construir uma simulação, o usuário deve adicionar ou estender as classes do *framework* de acordo com o protocolo específico e configuração avaliada, mas ainda utiliza a maior parte da funcionalidade do *framework* sem ter que reimplementar esta (por exemplo, envio de mensagens unicast e multicast).

4.2.1 API - *Application Program Interface*

Simplicidade de uso é um dos principais objetivos de projeto do SIMMCAST. O uso de módulos de processamento (no caso, *threads*) síncronos é encorajado pelo modelo de simulação baseado em processo. Toda a arquitetura se baseia em dez operações primitivas disponibilizadas ao usuário, e são classificadas como:

1. Primitivas de TRANSMISSÃO

send envia um pacote para o endereço destino, tanto unicast como multicast

2. Primitivas de RECEPÇÃO BLOQUEANTE E NÃO BLOQUEANTE

receive requisita a recepção de um pacote, bloqueando até o pacote ser disponibilizado

tryReceive tenta receber um pacote, retornando NULL se nenhum pacote estiver disponível

3. Primitivas de GERENCIAMENTO DE GRUPOS MULTICAST

join associa endereços de nodo à determinado grupo de multicast

leave remove endereços de nodo de determinado grupo de multicast

4. Primitivas de CONTROLE DE EVENTOS ASSÍNCRONOS

setTimer configura o temporizador para expirar em determinado tempo futuro

CancelTimer cancela um temporizador existente

onTimer método invocado quando o temporizador expira

5. Primitivas de CONTROLE DE PROCESSOS

sleep coloca a tarefa corrente para dormir por um tempo especificado

wakeUp acorda tarefa que pode estar dormindo

4.2.2 Metodologia de Simulação no SIMMCAST

Simulações com o SIMMCAST são desenvolvidas através da extensão do *framework*. Como já mencionado, o usuário deve estender os *hot spots* adicionando a lógica do protocolo desejado e a configuração que será avaliada. É criado um ambiente personalizado mantendo-se as funcionalidades do *framework*, não sendo necessário reescrever tais funcionalidades.

Blocos básicos de construção[2] são a chave da modularização da simulação e são especializados para fornecer adição de código através de herança, definindo a lógica do protocolo e outros comportamentos específicos. Através de composição, a descrição dos experimentos é feita pela combinação do conjunto de blocos.

No SIMMCAST uma simulação é descrita combinando-se um conjunto de blocos básicos, provendo código adicional quando necessário. Estes são constituídos de dois componentes principais: processos e filas. Processos são objetos ativos que correspondem às *threads* de execução. Eles adicionam e removem objetos das filas. Filas são usadas principalmente para modelar pacotes em trânsito.

A seguir, os blocos básicos do SIMMCAST são apresentados; para cada bloco básico, existe uma classe correspondente. Uma representação simplificada é incluída na Figura 4.1 ([44]).

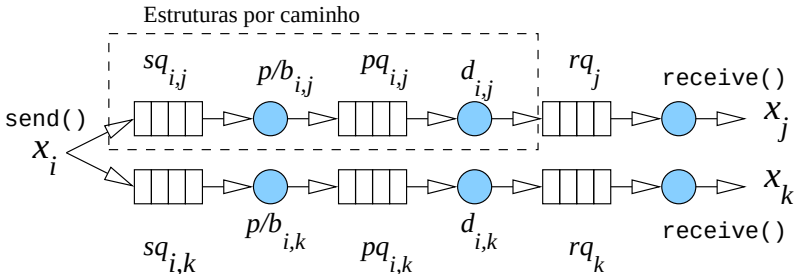


FIGURA 4.1 – Fluxo de Pacotes

Nodos — Class `Node` - Entidade fundamental da simulação.
Dependendo do nível de abstração desejado, nodos podem representar um agente de protocolo em um *host*, um roteador, ou uma das muitas entidades que interagem em um *host* ou roteador. Cada nodo x mantém três ou mais filas: *send queue:* uma fila de envio para cada caminho de saída, denotada como $sq_{x,y}$, para envios de x para y; *receive queue*: uma única fila de recebimento, denotada como rq_x, onde todos os pacotes que chegam a x são adicionados; *timer queue*: uma fila de temporizador, denotada como tq_x, para registrar futuros eventos assíncronos (não representada na Figura 4.1).

Caminhos — Class `Path` - Nodos são conectados por caminhos de pacotes. As propriedades do caminho são: banda, probabilidade de perda de pacote e

[2] building blocks

atraso de propagação. Todos os caminhos são unidirecionais, tal que o modelo tradicional de ligações físicas devem ser representados por dois caminhos. O caminho entre os nós x_i e x_j é representado pela fila $pq_{i,j}$ (*pq - path queue*), e ele armazena todos os pacotes em trânsito enviados de x_i para x_j. Dependendo do modelo representado, pode-se fazer com que caminhos no SIMMCAST representem um *link* físico, uma conexão que combine propriedades de rede de diversos *links* físicos, ou uma fila de mensagens local.

Grupo — Class `Group` - O conceito de grupo é fundamental para protocolos multicast. A participação em um grupo pode ser definida pelo usuário através do arquivo de descrição de simulação que descreve o experimento. Alternativamente, os nodos, através do código de protocolo associado, podem assinar ou deixar grupos dinamicamente, de acordo com a lógica do protocolo.

Rede — Class `Network` - Uma rede é uma combinação arbitrária de nodos e todos caminhos que os conectam. A arquitetura em camadas do SIMMCAST não impõe nenhum esquema de roteamento específico, para que diferentes tipos de esquemas possam ser livremente implementados estendendo a classe `Node`. Para um pacote ser transmitido do nodo x_i para o nodo x_j, um caminho deve ser criado conectando os dois (tal que, $\exists pq_{i,j}$). Da mesma forma, para x_i enviar um pacote para um grupo g , ele deve estar diretamente conectado com cada elemento x_j do grupo g (para cada $x_j \in g$, $\exists pq_{i,j}$). Estre modelo lógico, onde o caminho representa uma conexão entre dois nodos é adequado para representar simulações de sistemas distribuídos, não havendo, neste caso, roteamento explícito.

Pacotes — Class `Packet` - São a unidade de transmissão de dados entre dois ou mais nodos. A classe `Packet` contém os atributos mínimos necessários a um pacote no SIMMCAST, e para novos protocolos, herança é usada para definir novos tipos de pacotes.

Como já mencionado, o uso de simulação permite que protocolos sejam avaliados de acordo com um conjunto de métricas. A título de ilustração, são considerados os critérios típicos para avaliação de protocolos multicast confiáveis, *throughput* e custo de rede. Relembrando, *throughput* é usualmente definido como a quantidade efetiva de dados transmitidos sobre o tempo necessário para transmiti-los de maneira confiável, e custo de rede pode ser definido como a quantidade de banda necessária para completar (de forma confiável) a transmissão de todos os dados. Na comunicação unicast, esta é uma definição simples. Entretanto, é ligeiramente mais complexo avaliar o custo de rede quando se utiliza multicast, devido à replicação de pacotes através da árvore de distribuição. Para determinar o custo de transmissão, somam-se todos os pacotes, bits ou bytes transportados através de todos os caminhos (dados e controle) que constituem a topologia.

Além de métricas de saída, arquivos de traço são um recurso essencial de simuladores de protocolo. Eles permitem que o comportamento de um protocolo seja investigado de forma similar ao que ocorre com um *"sniffer"* em uma rede real. Como o

SIMMCAST é baseado em simulação discreta, ele permite que estes eventos sejam registrados em arquivos de traço. SIMMCAST considera a existência de apenas dois tipos gerais de eventos: inclusão e remoção de filas.

A Figura 4.1 ilustra as filas e tempos associados que um pacote deve passar enquanto é transmitido entre dois nodos diretamente conectados, x_i e x_j. O nodo x_i envia um pacote multicast para um grupo a que pertencem x_j e x_k; uma cópia do pacote é enfileirada em $sq_{i,j}$ e $sq_{i,k}$, existindo espaço em cada uma destas filas. O nodo x_i é bloqueado por um período igual a t_{send} , quando $t_{send} > 0$; o propósito de t_{send} é permitir ao usuário "transparentemente" limitar as taxas de transmissão. Um pacote em $sq_{i,j}$ mais cedo ou mais tarde chega ao início da fila, e então espera por um tempo igual a $p/b_{i,j}$, com p igual ao tamanho do pacote e $b_{i,j}$ à largura de banda do caminho de x_i para x_j. Após isso, o pacote deixa $sq_{i,j}$ e se junta à $pq_{i,j}$. O pacote fica na $pq_{i,j}$ por tempo $d_{i,j}$, onde $d_{i,j}$ é a latência de atraso do caminho de x_i para x_j. Mais tarde, o pacote é enfileirado em rq_j, caso exista espaço nesta fila. Após um tempo, o pacote chega ao início da fila rq_j, e a partir de então, no próximo `receive()` ou `tryReceive()` de uma *thread* em x_j, o pacote parte de rq_j. O tempo de serviço é ditado pela lógica do protocolo, mas não pode ser inferior a t_{recv}.

Para completar a especificação do nodo, existem dois blocos básicos que usualmente correspondem a recursos do sistema operacional: *threads* e temporizadores. *Threads* simplificam um protocolo porque elas permitem que o desenvolvedor modele a arquitetura como um conjunto de entidades síncronas mais simples que interagem entre si; por outro lado, *threads* podem ser disparadas dinamicamente para gerenciar mensagens de entrada (modelo com uma *thread* por mensagem).

No SIMMCAST cada processo é associado a uma *thread*. Eventos assíncronos podem ser implementados utilizando temporizadores. Existem muitos casos de eventos assíncronos em software de protocolos, sendo *timeouts* os exemplos mais comuns. *Timeouts* são usados, por exemplo, para detectar perda de pacotes. Além disso, *timeouts* permanentes e consecutivos permitem que um nodo detecte que ocorreu um particionamento na rede ocorreu. Temporizadores também podem ser usados para implementar comportamento periódico em protocolos.

4.3 Descrição da Implementação do Simulador SIMMCAST

A implementação do SIMMCAST é dividida em pacotes. Cada pacote contém um conjunto de classes inter-relacionadas que oferecem as características e comportamento mínimo necessário para descrever a simulação de protocolos multicast. A simulação é descrita através da extensão das classes abstratas (*hot spots*), acrescentando o comportamento desejado e particular para cada protocolo que se deseja simular.

Os principais pacotes que compõe o simulador SIMMCAST são citados a seguir e descritos nas próximas subseções:

- network;
- node;

- group;
- trace e
- script.

4.3.1 Pacote Network

Este é o principal pacote do SIMMCAST. Para realizar uma simulação, é necessário estender a classe abstrata `Network` (*hot spot*). A configuração e a topologia da rede são descritas em um arquivo de configuração que será interpretado pelo seu método `execute()`. Possui também classes que implementam a estrutura básica do mecanismo interno utilizado pelo simulador (filas) e a descrição necessária do elemento de comunicação do protocolo (pacote).

A Figura 4.9 apresenta o diagrama de classes do pacote Network, conforme notação UML ([23]).

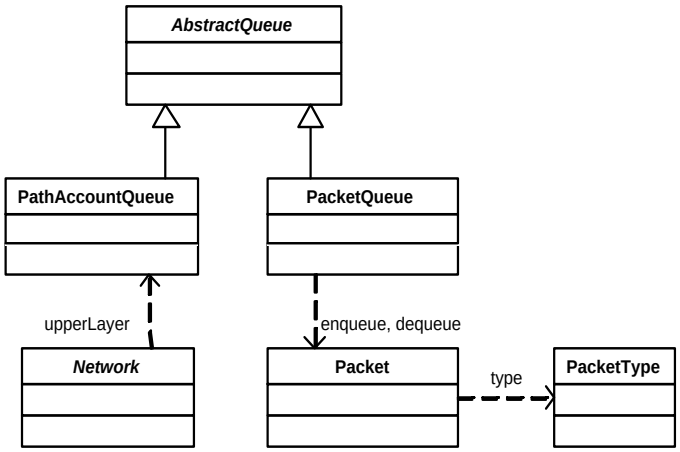


FIGURA 4.2 – Diagrama de Classes do Pacote Network

A seguir são apresentadas as descrições das classes do pacote Network. São elas: `Network`, `Packet`, `PacketType`, `AbstractQueue`, `PacketQueue` e `PathAccountQueue`.

`Network`

Classe abstrata que fornece o método `execute()` que inicializa a simulação. Controla o término da simulação através da contabilização das *threads* ativas: incrementa o contador cada vez que uma *thread* é inicializada, e decrementa o contador cada vez que uma *thread* é finalizada; se o contador for igual a zero então a simulação chegou ao fim. Contém os seguintes métodos globais:

- `setTrace()`: define o tipo de arquivo de traço que será gerado pelo simulador;
- `setGarbageCollectionInterval()`: define a periodicidade para executar o *garbage collection*;

- `simulationTime()`: obtém o tempo atual da simulação;
- `getNodeByID()`: obtém o descritor (objeto) do nó pelo identificador;
- `obtainUnicastAddress()`: obtém um endereço unicast;
- `obtainMulticastAddress()`: obtém um endereço multicast.

`Packet`

Classe que define um pacote. Possui os atributos básicos necessários para um pacote da simulação como: `from`, `to`, `type` e `size`. Esta classe pode ser estendida definindo-se um pacote específico para a simulação.

`PacketType`

O pacote utilizado na simulação é representado pela classe `Packet`. A classe `Packet` contém um atributo para identificar o tipo do pacote - `type`. Este atributo é um objeto do tipo `PacketType`. Originalmente o SIMMCAST possui a classe `PacketType` vazia, ou seja, nenhuma definição especial foi feita para o tipo do pacote, e nestes casos, o tipo de pacote pode ser `null`. Caso seja necessário diferenciar os pacotes na simulação do protocolo, pode-se estender esta classe definindo as características necessárias para a classificação dos pacotes por "tipo".

`AbstractQueue`

Classe abstrata que oferece um conjunto de operações básicas para manipulação e gerenciamento das filas geradas pelo simulador. É a classe ancestral das classes `PathAccountQueue` e `PacketQueue`.

`PacketQueue`

Implementa a fila de pacotes existentes nos caminhos entes os nós da rede. Possui a política FIFO[3] e o seu tamanho pode ser limitado. As filas implementadas por esta classe são: `SQ-SendQueue`, `RQ-ReceiveQueue` e `PQ-PathQueue`.

`PathAccountQueue`

Não é realmente a implementação de uma fila, somente contabiliza a quantidade de pacotes colocados nas filas. Possui os métodos `enqueue()` e `dequeue()` que gerenciam a contabilização.

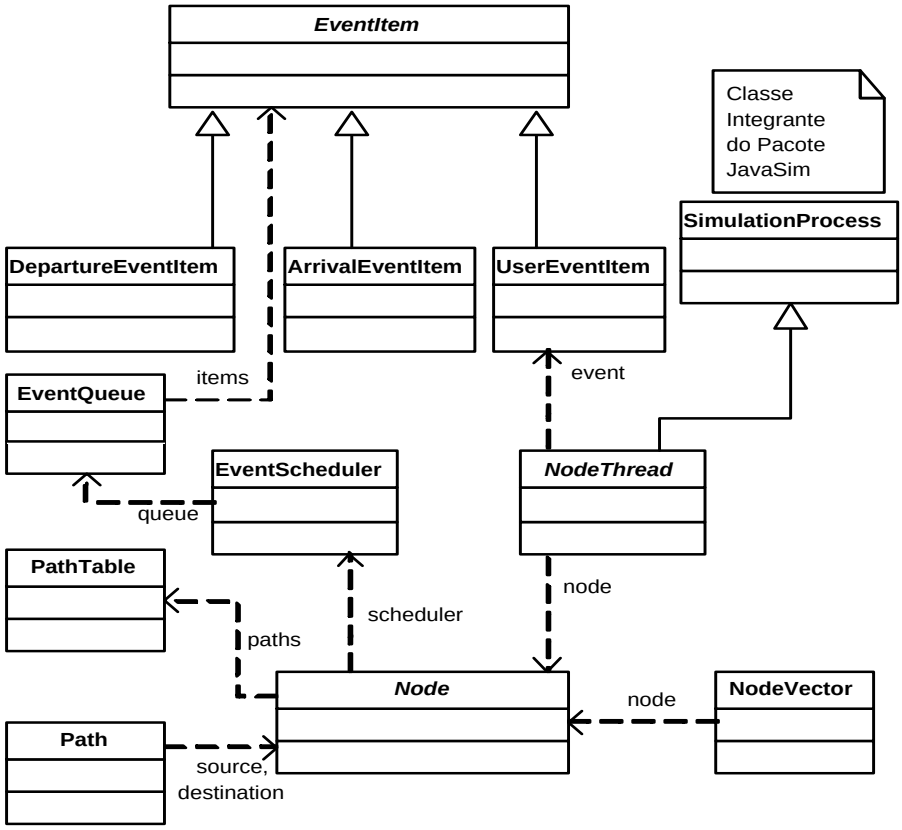


FIGURA 4.3 – Diagrama de Classes do Pacote Node

4.3.2 Pacote Node

Este pacote completa, com duas classes abstratas `Node` e `NodeThread` (*hot spots*), o conjunto de classes mínimas necessárias para descrever a simulação. A Figura 4.3 apresenta o diagrama de classes do pacote Node. São elas: Node, NodeThread, EventQueue, EventItem, ArrivalEventItem, DepartureEventItem, UserEventItem, EventScheduler, NodeVector, Path e PathTable e são detalhadas na seqüência. O diagrama UML do pacote Node será posteriormente detalhado - com a inclusão de atributos e métodos globais - na Seção 5.3.2.

`Node`

Esta classe abstrata descreve as características mínimas necessárias para representar um nodo da rede que se deseja simular. Possui o método `addPath()` para definir os caminhos conectados a este nodo. Cada nodo da rede deve conter pelo menos uma *thread*, que é o objeto ativo do nodo, de modo que as tarefas que ele precisar realizar ao receber e enviar pacotes são executas pela *thread*. Esta classe contém os objetos `scheduler`, `paths` e `receiverQueue`, que respectivamente representam, o escalonador das tarefas do nodo, a tabela de caminhos existentes neste nodo e a fila de pacotes recebidos por ele.

[3]First In First Out

`NodeThread`

Classe abstrata que deve ser estendida pelo usuário para criar/definir o comportamento de um nodo. A classe `NodeThread` implementa o objeto ativo existente em cada nodo. Possui os métodos `send()` para enviar pacotes, `receive()` e `tryReceive()` para receber pacotes e `sleep()` e `wakeUp()` para bloquear e desbloquear a *thread*. Como esta classe implementa uma *thread*, há possibilidade de criar nós com múltiplas *threads*, ou seja, os nodos podem conter várias *threads* para tratar e gerenciar o comportamento desejado.

`EventQueue`

Define a fila de eventos que são gerenciados pelo escalonador (classe `EventScheduler`). Possui o método `enqueue()` para inserir eventos, o método `dequeueHead()` para retirar os eventos da fila e o método `cancelUserEvent()` para cancelar um evento do usuário.

`EventItem`

Classe abstrata para a definição dos tipos de eventos que serão gerenciados pela `EventQueue`. É a classe ancestral de `ArrivalEventItem`, `DepartureEventItem` e `UserEventItem`.

`ArrivalEventItem`

Classe que define os eventos de chegada de pacotes no nodo.

`DepartureEventItem`

Classe que define os eventos de saída de pacotes do nodo.

`UserEventItem`

Classe que define um evento personalizado gerado e controlado pelo usuário. A classe `NodeThread` possui o método `onTimer()` que é disparado sempre que o evento for ser executado.

`EventScheduler`

É uma *thread* que controla e escalona os eventos gerados pelo simulador (saída e chegada de pacotes e eventos do usuário). É criada internamente pelo simulador para cada nodo da simulação.

`NodeVector`

Tabela que contém os nodos existentes na simulação.

`Path`

Path representa o fluxo de pacotes entre dois nodos. São usados para conectar estes nodos e podem representar uma fila de pacotes, um *link* físico ou um caminho lógico. Todos os caminhos entre dois nodos *hosts* são unidirecionais, de modo que para modelar um caminho físico bi-direcional, dois *links* devem ser definidos.

`PathTable`

Tabela que armazena os caminhos existentes entre os nodos da simulação.

4.3.3 Pacote Group

Este pacote disponibiliza as classes que definem os grupos multicast existentes na simulação. É composto apenas por duas classes, `Group` e `GroupTable`, como representado na Figura 4.4.

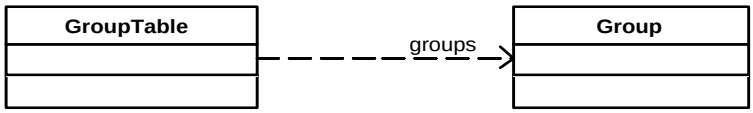


FIGURA 4.4 – Diagrama de Classes do Pacote Group

`Group`

Representa os grupos multicast da simulação. Normalmente definidos pelo arquivo de definição da simulação. Possui o método `join()` para associar nodos ao grupo, o método `leave()` para desassociar nodos ao grupo e o método `size()` para retornar o tamanho do grupo.

`GroupTable`

Tabela que armazena os grupos existentes na simulação.

4.3.4 Pacote Trace

O pacote Trace é composto por classes que gerenciam os arquivos de traço da simulação. Na seqüência é apresentada a descrição das classes existentes no pacote. São elas: `TraceGenerator, NullTraceGenerator, ImplosionTraceGenerator`, `File TraceGenerator`, `SimmcastTraceGenerator` e `NamTraceGenerator`. A Figura 4.5 mostra o diagrama de classes do pacote Trace.

`TraceGenerator`

Classe abstrata que descreve os métodos existentes que geram entradas no arquivo de traço. É a classe ancestral de `NullTraceGenerator`, `ImplosionTraceGenerator` e `FileTraceGenerator.` Possui os seguintes métodos:

- `message()`: adiciona uma mensagem qualquer ao arquivo de traço;
- `error()`: adiciona uma mensagem qualquer ao arquivo de traço;
- `nodeMessage()`: adiciona uma mensagem qualquer incluindo o identificador do nodo;
- `nodeError()`: adiciona uma mensagem de erro incluindo o identificador do nodo;

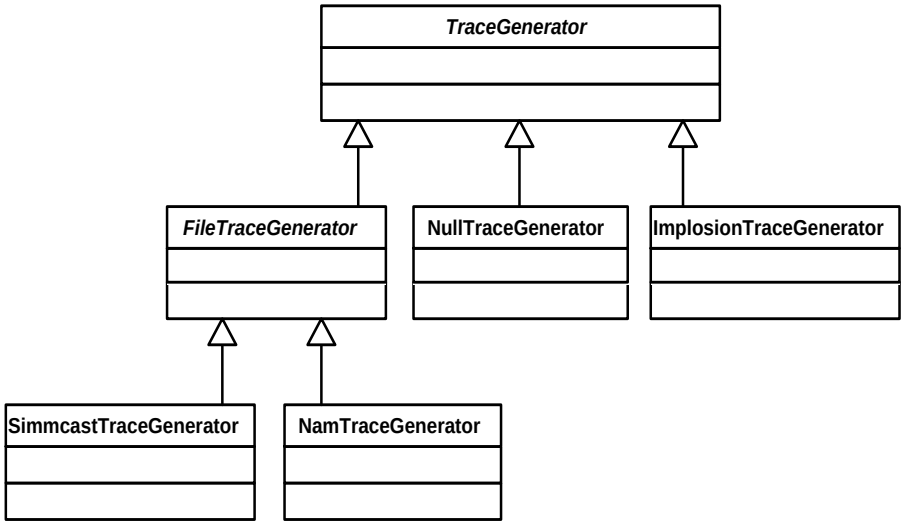


FIGURA 4.5 – Diagrama de Classes do Pacote Trace

- `move()`: adiciona uma mensagem de transferência de pacotes entre as filas;
- `loss()`: adiciona uma mensagem quando há perda de pacotes;
- `node()`: adiciona uma mensagem quando um nodo é criado;
- `path()`: adiciona uma mensagem quando um caminho é criado.

`NullTraceGenerator`

Classe estendida de `TraceGenerator` que anula e não implementa nenhum método. Deve ser utilizado quando não se deseja gerar traços na simulação.

`ImplosionTraceGenerator`

Classe que gera um arquivo de traço com as perdas geradas devido à implosão, ou seja, contabiliza a quantidade de implosões de *acks*.

`FileTraceGenerator`

Classe que estende a `TraceGenerator` acrescentando o controle do arquivo de traço. Classe ancestral das classes `SimmcastTraceGenerator`.

`SimmcastTraceGenerator`

Classe que gera um arquivo de traço no formato nativo do simulador armazenando as mensagens em um arquivo.

`NamTraceGenerator`

Classe que gera um arquivo de traço no formato utilizado pelo animador nam, visualizador padrão do simulador ns (vide Seção 3.2).

4.3.5 Pacote Script

Pacote do simulador que possui a responsabilidade de interpretar e executar o arquivo que define a simulação. Este arquivo é um texto que possibilita descrever a topologia da rede, definir os grupos, associar os nós aos grupos e definir o tipo de traço que será utilizado pelo simulador, onde cada linha deve conter no máximo um comando. Como pode ser verificado na Figura 4.6, este pacote é composto apenas por duas classes: `ScriptParser` e `MacroPreprocessor`.

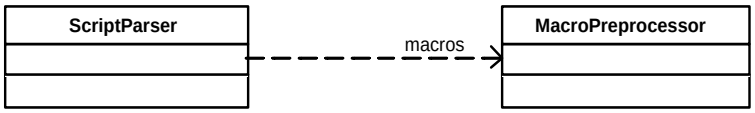


FIGURA 4.6 – Diagrama de Classes do Pacote Script

`ScriptParser`

Classe que interpreta e executa o arquivo que define a simulação.

`MacroPreprocessor`

Classe que interpreta e executa as macros contidas no arquivo de definição da simulação.

4.4 Criando uma Simulação com SIMMCAST

Para executar uma simulação, é necessário especificar:

- **o protocolo a ser simulado**: através da combinação de blocos básicos existentes, especializando os *hot spots*;
- **a topologia da rede:** instâncias dos nodos e a definição de sua conectividade; e,
- **a topologia do protocolo:** como especificar a alocação de agentes na rede previamente definida.

Os dois últimos passos são feitos preparando um arquivo de descrição de simulação, sem a necessidade de recompilação. A seguir é apresentado um breve tutorial de como desenvolver uma simulação com o SIMMCAST, baseado em [58].

4.4.1 Definição de Nodos

Cada tipo de entidade na simulação é descrito por um objeto nodo (uma subclasse de `Node`). O comportamento dos nodos será dado pelas *threads* (subclasses de `NodeThread`) que elas contém. No início da simulação, o SIMMCAST chama o método `begin()` do nodo criado. Neste método, são criados os métodos do protocolo a ser simulado e estes são iniciados pela chamada do método `launch()`.

A Figura 4.7 mostra um esqueleto que representa o procedimento apresentado para nodos transmissor e receptor.

```
public class SourceNode extends Node {
  // atributos para serem compartilhados por todas as threads
  int sourceNodeAttribute;
  // torna publico para ser acessado pelo arquivo de simulação
  public void setSourceNodeAttribute(int value) {
    sourceNodeAttribute = value;
  }
  // cria e inicializa as threads
  public void begin() {
    SourceThread thread = new SourceThread(this);
    thread.launch();
  }
}


public class ReceiverNode extends Node {
  // atributos para serem compartilhados por todas as threads
  int receiverNodeAttribute;
  // torna publico para ser acessado pelo arquivo de simulação
  public void setReceiverNodeAttribute(int value) {
    receiverNodeAttribute = value;
  }
  // cria e inicializa as threads
  public void begin() {
    ReceiverThread thread = new ReceiverThread(this);
    thread.launch();
  }
}
```

FIGURA 4.7 – Definição dos Nodos

4.4.2 Definição de *Threads*

Os objetos *thread* contém a descrição do comportamento dos nodos da simulação, como, por exemplo, a lógica de um protocolo. Da mesma forma que o método `run()` das *threads* Java, SIMMCAST contém o método `execute()` que permite incluir código a ser executado. Deve ser observado que, como o método construtor das classe não é herdado por *default*, é necessário redefinir o construtor da classe `NodeThread`. Isto pode ser observado na Figura 4.8, que mostra a definição das *threads* para nodos transmissor e receptor, bem como atributos para armazenar o identificador do nodo.

```
class SourceThread extends NodeThread {
  SourceNode source;
  // salva o identificador do nodo
  public SourceThread (Node node_) {
    super (node_);
    source = (SourceNode)node_;
  }
  public void execute() {
    // incluir lógica da simulação do nodo transmissor
    // baseado em primitivas send(), receive(), etc.
  }
}


class ReceiverThread extends NodeThread {
  ReceiverNode receiver;
  // salva o identificador do nodo
  public ReceiverThread (Node node_) {
    super (node_);
    receiver = (ReceiverNode)node_;
  }
  public void execute() {
    // incluir lógica da simulação do nodo receptor
    // baseado em primitivas send(), receive(), etc.
  }
}
```

FIGURA 4.8 – Definição das *Threads*

4.4.3 Definição da Rede

Quando da definição do objeto `network` da simulação, tem-se este como o lugar indicado para as definições de uso global, como por exemplo *PacketType*, se for necessário diferenciar categorias de pacotes; se isto não for necessário, pode-se utilizar *PacketType.DEFAULT* na operação de `send()`. Um exemplo é mostrado na Figura 4.9.

```
class SimulationNetwork extends Network {
  // tipo de pacote
  public static PacketType PACKET_TYPE();
}
```

FIGURA 4.9 – Definição da Rede

4.4.4 Definição da Classe *Main*

Na definição da classe main da simulação devem ser incluídos: a criação do objeto `network`, passando um *string* contendo o nome do arquivo de simulação; e uma chamada do método `execute()`, do Java, até que a simulação termine. Isto é mostrado na Figura 4.10.

```
public class Test {
  public static void main (String S[]) {
    SimulationNetwork network = new SimulationNetwork();
    if (S.lenght < 1) {
      System.err.println(¨Especifique o arquivo de simulação!¨);
    }
    network.execute (S[1]);
    try {
      network.join();
    } catch (Exception e) { System.out.println(e); }
  }
}
```

FIGURA 4.10 – Definição da Classe *Main*

4.4.5 Definição do Arquivo de Simulação

O arquivo de simulação é tido como um arquivo texto que contém informações relativas à simulação. Nele devem ser criados os nodos, descritas as conexões entre eles, ou seja, a topologia da rede propriamente dita, e a parametrização do experimento. Um exemplo de arquivo de definição da simulação é mostrado na Figura 4.11 que mostra o arquivo *"modelo.sim"*.

A Figura 4.12 exibe a topologia da rede (trivial, por questões didáticas) criada pelo arquivo de simulação *"modelo.sim"*.

4.4.6 Executando a Simulação

Após escritos os arquivos que compõem a simulação, esta é iniciada com o comando: `java Test modelo.sim` .

```
# Cria nodos...
new S SourceNode
new R[1..10] ReceiverNode
# Define um grupo multicast
new G simmcast.group.Group
G join R[1..10]
# Relaciona atributos definidos pelo usuário
S setSourceNodeAttribute 12345
R1 setReceiverNodeAttibute 67890
# Define caminhos com buffer = 10, largura de banda = 1000,
# taxa de perda de pacotes = 1% e
# atraso descrito conforme uma distribuição normal.
# Caminhos são unidirecionais, de modo que para se descrever
# um caminho bi-direcional é descrito um par de caminhos
new normalDist arjuna.JavaSim.Distributions.NormalStream 50 10
s addPath R[1..10] 10 1000 normalDist 0.01
R[1..10] addPath s 10 1000 normalDist 0.01
```

FIGURA 4.11 – Definição do Arquivo de Simulação

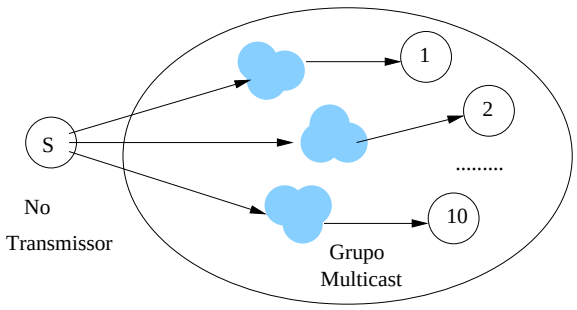


FIGURA 4.12 – Topologia da Rede Criada pelo Arquivo "*modelo.sim*"

Diferentes níveis de representação da rede são possíveis. No nível mais abstrato, como no exemplo apresentado, a rede é um conjunto de conexões diretas (caminhos) ponto-a-ponto entre os nodos. Vários destes modelos de topologia podem ser considerados. Um maior nível de detalhe pode ser obtido fazendo parte dos nodos agirem como roteadores. Para tanto, alterações na arquitetura atual do SIMMCAST foram propostas e implementadas, conforme apresentado no capítulo a seguir.

Capítulo 5

ROTEAMENTO MULTICAST NO SIMMCAST

Uma simulação no SIMMCAST abrange um largo espectro de abstração, variando o significado atribuído a um nodo e suas filas. Em um extremo, uma simulação pode contar com um conjunto de processos de aplicação conectados através de caminhos lógicos, onde o conjunto de conexões entre nodos formará a topologia lógica empregada para simular um sistema distribuído. No outro extremo, uma simulação pode ser detalhada a ponto de representar interações entre camadas de protocolos em múltiplos elementos de rede (*hosts*, roteadores, comutadores).

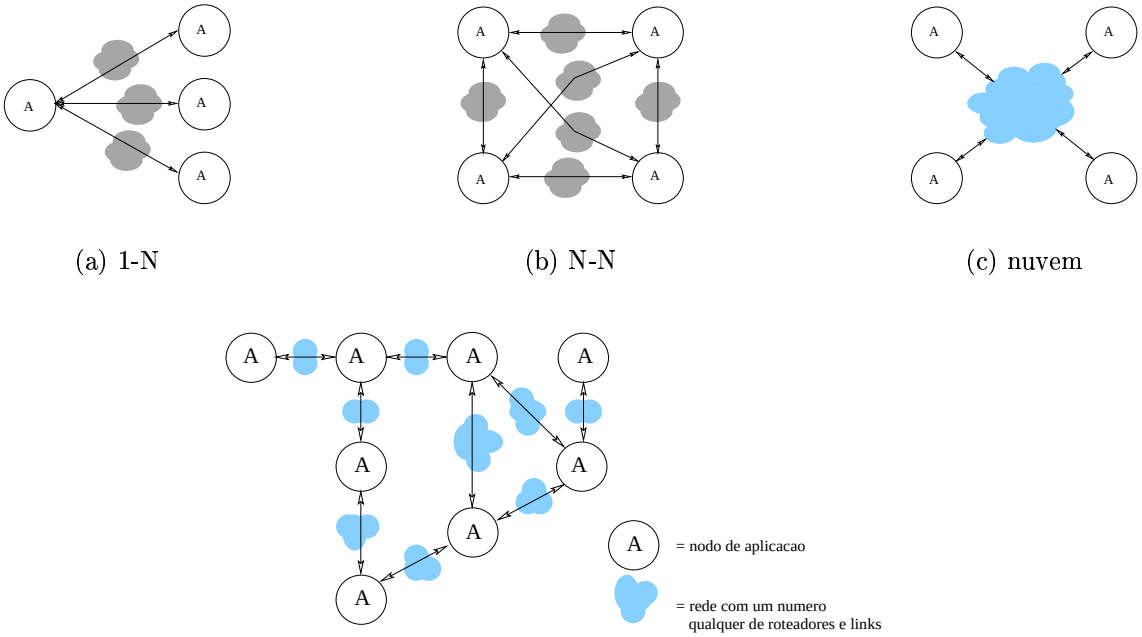
A arquitetura do SIMMCAST descrita no capítulo anterior não conta com lógica de roteamento; nela, caminhos de pacotes representam conexões lógicas individuais entre dois nodos, não havendo compartilhamento algum entre caminhos diferentes. Um caminho modela de forma abstrata as propriedades relativas a uma seqüência de elementos de rede contidas entre dois nodos, fazendo com que cada pacote experimente um atraso diferente. Por exemplo, um canal confiável FIFO, similarmente ao "serviço" TCP, pode ser implementado através deste modelo. Nesta arquitetura, nodos podem representar diferentes níveis de abstração, de um processo pertencente a uma aplicação distribuída a elementos de rede, tal como roteadores. No primeiro caso, caminhos tipicamente teriam atrasos aleatórios, mas sem perdas, enquanto no segundo teriam atraso fixo mas probabilidade de perda não nula.

Conexões lógicas são muito abstratas para a simulação de protocolos (com funcionalidade ao nível) de transporte. Nestes casos, e em tantos outros, é necessário incluir no modelo a topologia física da rede. Para se usar uma topologia física arbitrária, é necessário que pacotes sejam encaminhados entre origem e destinos através de múltiplos nodos roteadores. As rotas a serem seguidas dependerão dos algoritmos de roteamento implementados nos roteadores. A Seção 5.1 apresenta alguns exemplos de topologia de rede a serem utilizadas em simulações com o SIMMCAST. A Seção 5.2 descreve a nova arquitetura do simulador com suporte a roteamento, a Seção 5.3 a descrição das classes envolvidas na implementação do novo modelo enquanto que a Seção 5.4 tece comentários relativos ao novo modelo. A arquitetura apresentada é uma extenção do SIMMCAST para permitir experimentos que incluam roteamento multicast, servindo tanto a protocolos que necessitam uma topologia com roteamento subjacente (experimentos em nível de transporte/aplicação) como

para o estudo e avaliação de protocolos de roteamento em si (experimentos em nível de rede).

5.1 Topologias no SIMMCAST

Existe uma gama de experimentos de simulação que podem ser realizados utilizando modelos sem roteamento. Alguns exemplos de topologias são ilustrados na Figura 5.1 [45] e comentados a seguir.



(a) 1-N (b) N-N (c) nuvem

(d) grafo

FIGURA 5.1 – Exemplos de Modelos sem Roteamento

O caso (a) na Figura 5.1 representa uma comunicação multicast um-para-vários onde existe um caminho lógico bi-direcional um-para-um entre o transmissor e cada um dos receptores. Uma variação deste modelo é apresentada no caso (b), que ilustra uma comunicação multicast vários-para-vários. Em ambos os casos, os caminhos lógicos são responsáveis por modelar probabilisticamente atrasos e perdas de pacotes. A completa abstração da camada de rede provida por estes modelos se adequa bem ao estudo de protocolos de nível mais alto, como por exemplo, algoritmos distribuídos para eleição ou formação de árvore [39].

Os caminhos lógicos acima são independentes entre si, e portanto não modelam compartilhamento do meio. Um modelo de topologia que mantém a abstração de caminhos lógicos e é capaz de representar este compartilhamento é ilustrado no caso (c). Tal é obtido através de um nodo central que age como "nuvem". Nele, uma matriz descreve as propriedades de comunicação entre cada um dos pares de nodos e é responsável pela implementação de atrasos aleatórios na distribuição de pacotes.

Para cada nodo, existe na nuvem uma fila de saída associada, por onde passam todos os pacotes destinados ao nodo em questão. Esta fila tem dois propósitos: adicionar atrasos de enfileiramento e perdas devido a estouro de *buffer*. Com este modelo, torna-se possível a representação em alto nível de uma rede apenas em função de atrasos e perdas fim-a-fim.

Os casos (a), (b) e (c) utilizam modelos abstratos de topologia. Em determinadas situações, será necessário precisá-la, seja em função de regras lógicas ditadas pela aplicação, ou em função da interconexão física entre os nodos. O caso (d) ilustra estas situações, onde todos os nodos são processos de aplicação ou agentes de transporte, mas há restrições na interação entre os mesmos (especificadas na figura através de um grafo).

Em [6, 44], são descritos casos baseados em modelos abstratos de topologias. Ao considerar simulações com redes físicas arbitrárias, um simulador deve apresentar as seguintes características:

- independência de roteamento, ou seja, a lógica dos protocolos ou sistemas que são objeto de estudo não são alterados em função do roteamento, permitindo usar abordagem incremental;
- separação clara entre a camada (protocolo ou sistema) sendo simulada e a abstração de rede subjacente;
- utilização de topologias físicas arbitrárias, incluindo anel uni/bi-direcional, estrela, árvore, barramento, ou a combinação destas em uma outra topologia;
- geração de traços em diferentes níveis de abstração, permitindo visualizar interações entre elementos do protocolo ou sistema.

O suporte a roteamento, foco deste trabalho, expande as capacidades do ambiente de simulação, permitindo:

- importação de topologias descritas em arquivos e criadas por geradores automáticos de topologias, tal como o gt-itm - *Georgia Tech Internetwork Topology Models* ([24]);
- dispor de diferentes lógicas de roteamento multicast intra e inter-domínio, como árvore baseada em fonte ou compatilhada, e protocolos correspondentes;
- utilização de topologias dinâmicas com modelagem de falhas (temporárias ou permanentes) de *hosts*, roteadores e *links*, com inclusão, remoção e relocação dinâmicas de *hosts* e *links*.

Em função desses requisitos, explorou-se a extensibilidade da arquitetura do SIMMCAST, criando-se blocos derivados a partir dos blocos básicos de construção providos pelo *framework*, os quais são discutidos nas próximas seções.

5.2 Arquitetura do SIMMCAST com Suporte a Roteamento Multicast

A arquitetura do SIMMCAST foi modificada através da implementação de novos blocos de construção. Enquanto topologias lógicas são compostas de nodos aplicativos e caminhos, topologias físicas são compostas de *hosts*, roteadores e *links*. Refletindo isto, blocos de construção do tipo *path* assumem o papel de *links*, e surgem dois novos blocos de construção, derivados de *node:*

- *host*, e
- *router*.

Estes blocos são fundamentais para o SIMMCAST, à medida que permitem a execução de simulações mais complexas com protocolos de transporte, incluindo redes com topologias físicas arbitrárias. Como esperado, instâncias do bloco *host* detêm a lógica do protocolo ou aplicação sendo estudada, enquanto instâncias do bloco de construção *router* são responsáveis pelo encaminhamento e distribuição de pacotes. Um bloco de construção *host* é um nodo que possuirá apenas uma conexão bidirecional (isto é, dois caminhos, um de entrada e um de saída, conectados com um mesmo *router*). Um nodo *router* poderá possuir um número arbitrário de conexões de entrada e de saída.

É possível mapear todos os tipos básicos de topologia (anel, estrela, árvore e barramento) e a combinação destes utilizando *hosts*, *routers* e *paths*. Por exemplo, em uma topologia em árvore, a raiz e as folhas (fonte e receptores) são representadas por *hosts*, enquanto os nodos internos correspondem a *routers*. Um exemplo dessa topologia é ilustrado na Figura 5.2. A árvore indicada na figura é utilizada na distribuição de pacotes quando "F" transmite um pacote a um grupo contendo os receptores 1 a 7. Os roteadores 12 e 13 não estão marcados porque não tomam parte dessa árvore de caminho mais curto enraizada em "F". Similarmente, estão marcados apenas os caminhos que fazem parte da árvore.

No *framework*, os blocos de construção *host* e *router* correspondem às classes `HostNode` e `RouterNode`, subclasses de `Node`. Refletindo as características apontadas anteriormente, no `HostNode` o método `send()` passa a encaminhar toda mensagem enviada para o roteador associado, independemente do destinatário informado. O `RouterNode` estende o `Node` adicionando múltiplas filas de recebimento (rq), uma por *link* de entrada, segundo modelo de filas tipicamente encontrado em roteadores ([36]), além de uma fila de envio para cada *link*, já parte da definição de `Node`. Objetos `NodeThread` são usados para representar no `HostNode` a lógica de aplicação e no `RouterNode` o protocolo de roteamento.

Pacotes são encaminhados por roteadores de acordo com uma tabela de roteamento. Uma tabela de roteamento de dados representa uma abstração lógica da infra-estrutura de distribuição de pacotes unicast e multicast. Ela mapeia endereços a interfaces de saídas de dados e endereços do próximo destino.

A tabela de roteamento pode ser preenchida de forma estática (através do conhecimento da topologia da rede como um todo) ou dinâmica (através de um protocolo de roteamento). Estas duas abordagens são consideradas a seguir.

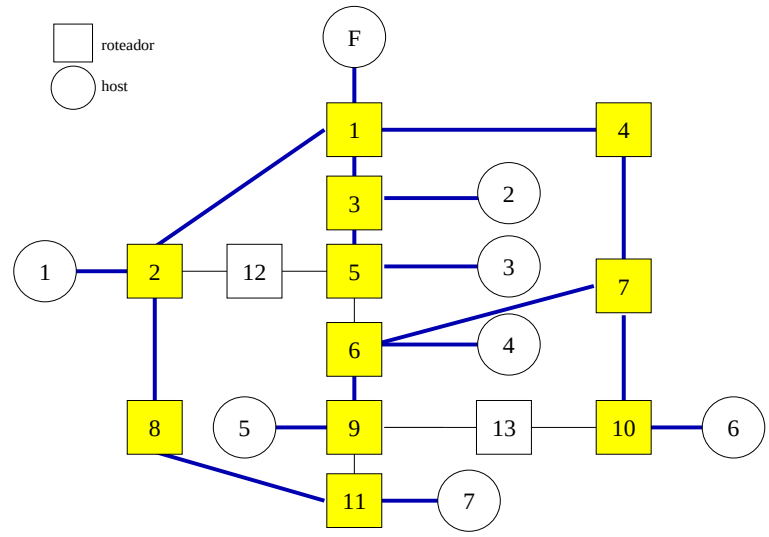


FIGURA 5.2 – Exemplo de Simulação em Topologia Arbitrária

5.2.1 Roteamento Estático

Roteamento estático pode ser utilizado em simulações onde as interações entre os roteadores para execução dos algoritmos de roteamento podem ser abstraídas. Neste modelo, a topologia da rede é acessível por todos os nodos roteadores, de modo que cada nodo pode construir sua própria tabela de roteamento utilizando um algoritmo de composição de árvore, que pode ser definido pelo usuário. O SIMMCAST implementa como padrão o algoritmo Dijkstra baseado em árvore de caminhos mais curtos[1] (vide Seção 2.1), mas permite que outros esquemas também sejam utilizados.

No modelo de roteamento estático, `DefaultRouterNode` (uma subclasse de `RouterNode`), contém uma ou mais *threads* (objetos `NodeThread`) responsáveis por consultar a tabela de roteamento e encaminhar os pacotes recebidos.

Na implementação do roteamento estático no SIMMCAST foi preterido o uso de uma tabela global de roteamento para todos os nodos, em prol de uma tabela de roteamento por nodo. Para a construção de sua tabela, que ocorre sob demanda, um nodo possui acesso (conhecimento) à topologia completa da rede. Isto se deu porque o uso de uma tabela global, que congrega o conjunto necessário de árvores, traz implicações como consumo excessivo de recursos de memória no roteador para armazenar a tabela, uma vez que o tamanho deste dependende do número de nodos, do número de grupos existentes e do número de fontes (em caso de árvore baseada na fonte[2] ; tal problema já foi identificado em [54]).

No entanto, em qualquer das abordagens, o tempo de convergência (isto é, o tempo gasto até que todos os nodos possuam suas tabelas de roteamento acertadas) é zero, uma vez que a tabela é montada antes do início do experimento de simulação, ou sob ordem da aplicação. A ocorrência de falhas de *links* e nodos, uma vez implementadas, refletirão imediatamente na tabela e, desta forma, percebida por todos os nodos ao mesmo tempo.

[1] shortest path tree
[2] source-based tree

5.2.2 Roteamento Dinâmico

O roteamento estático descrito na seção anterior é suficiente para grande parcela das simulações, onde os detalhes do funcionamento dos protocolos de roteamento são ignorados. No entanto, em certos casos, particularmente na investigação de protocolos e algoritmos de roteamento multicast, o objeto de estudo é a camada de rede, e portanto passa a ser necessário executar, além do protocolo de aplicação, o algoritmo de roteamento.

O roteamento dinâmico permite uma visão mais real e detalhada da rede. Ele pode ser utilizado tanto para execução de simulações com menor nível de abstração, como para estudo e análise de protocolos de roteamento multicast ([11]). Nestas situações, as métricas tipicamente utilizadas são tempo e custo até a convergência, atraso médio entre fonte e receptores e custo global em pacotes para uma transmissão multicast ([12]).

O suporte do SIMMCAST a estas métricas se dá no arquivo de traço através da identificação de classes de pacotes e eventos relevantes, como por exemplo a alteração de propriedades no *link*. A informação gerada no traço é extensível pelo usuário; o mesmo pode estender a classe SimmcastTraceGenerator, por exemplo (vide Seção 4.3.4), para adicionar ao traço dados específicos do protocolo desenvolvido.

No modelo de roteamento dinâmico, subclasses de `RouterNode` contém uma ou mais *threads* (objetos `NodeThread`) responsáveis por executar o algoritmo de roteamento. Cada subclasse que implementa algum protocolo de roteamento contém sua própria tabela (de roteamento), e é responsável por mantê-la atualizada. Isto é feito através de troca de mensagens de controle entre os nodos roteadores.

O uso deste modelo, mais realístico que o roteamento estático, impacta na aplicação ou protocolo de transporte através do acréscimo de tráfego na rede, causado tanto pelas mensagens de controle como pelo encaminhamento indevido de mensagens. Este último ocorre devido ao tempo necessário até a convergência da rede. Por exemplo, quando um *host* deixa um grupo, pacotes continuam sendo encaminhados a ele até que os devidos roteadores sejam notificados. Isto dependerá da topologia da árvore e da composição do grupo.

5.2.3 Inclusão Incremental de Roteamento

Os modelos de roteamento desenvolvidos para o SIMMCAST e apresentados nas seções anteriores foram concebidos de acordo com a filosofia incremental (evolução do abstrato para o detalhado) que rege o simulador. Por exemplo, em simulações de sistemas distribuídos baseados em comunicação em grupo e protocolos de multicast confiável em nível de transporte, pode-se tirar proveito da compatibilidade entre as diferentes abordagens providas pelo modelo de blocos *host* e *router*: sem roteamento, com roteamento estático e com roteamento dinâmico. Isto permite uma evolução gradual do experimento, exceto quando o mesmo envolve a investigação de protocolos de roteamento: neste caso, a lógica do objeto de estudo da simulação

reside no próprio `RouterNode`, ao invés de em `HostNode`. Similarmente, [32] apresenta uma técnica formal de construção incremental de especificação de sistemas e descrição e prova de algoritmos.

No SIMMCAST, a abordagem incremental é possível uma vez que as mesmas *threads* usadas pelos objetos `Node` podem ser embutidas em um objeto `HostNode`, de modo a iniciar a modelagem do protocolo utilizando um maior grau de abstração. Posteriormente, este mesmo protocolo poderá então ser submetido a uma rede com roteamento estático ou dinâmico, utilizando as mesmas *threads* em objetos `HostNode` que poderão interagir com objetos derivados de `RouterNode`. Desta forma, o experimento pode, sem esforço, utilizar tanto um dos modelos abstratos na simulação como uma topologia física arbitrária.

5.3 Implementação do Suporte a Roteamento no SIMMCAST

A implementação do roteamento foi baseada nos conceitos de *forwarding* e *routing* de pacotes ([48]). Tem-se por *forwarding* ou encaminhamento como a movimentação de pacotes de uma interface de entrada para uma interface de saída de acordo com uma decisão de encaminhamento baseada em uma tabela de rotas; e por *routing,* o processo de obter e manter atualizadas informações de roteamento nas tabelas armazenadas nos roteadores.

A implementação do suporte a roteamento no SIMMCAST impacta na inclusão de classes nos pacotes Node e Network, bem como na criação de um novo pacote denominado Route. Assim, os pacotes que compõem a nova arquitetura do simulador SIMMCAST são apresentados a seguir e descritos nas próximas subseções:

- network;
- node;
- route.

Os diagramas UML representam a nova composição dos pacotes Node e Network, onde as novas classes são apresentadas com fundo escuro, bem como as classes integrantes do pacote Route e são descritas nas próximas seções. Os diagramas UML para estes pacotes explicitam os métodos contidos em cada classe, de modo a facilitar a visualização da implementação da nova arquitetura.

5.3.1 Pacote Network

O pacote Network foi incrementado de três novas classes: `RouterPacket`, `QueueAble` e `PacketMultiQueue`, as quais são explicitadas a seguir. A Figura 5.3 apresenta o novo diagrama de classes do pacote `Network`.

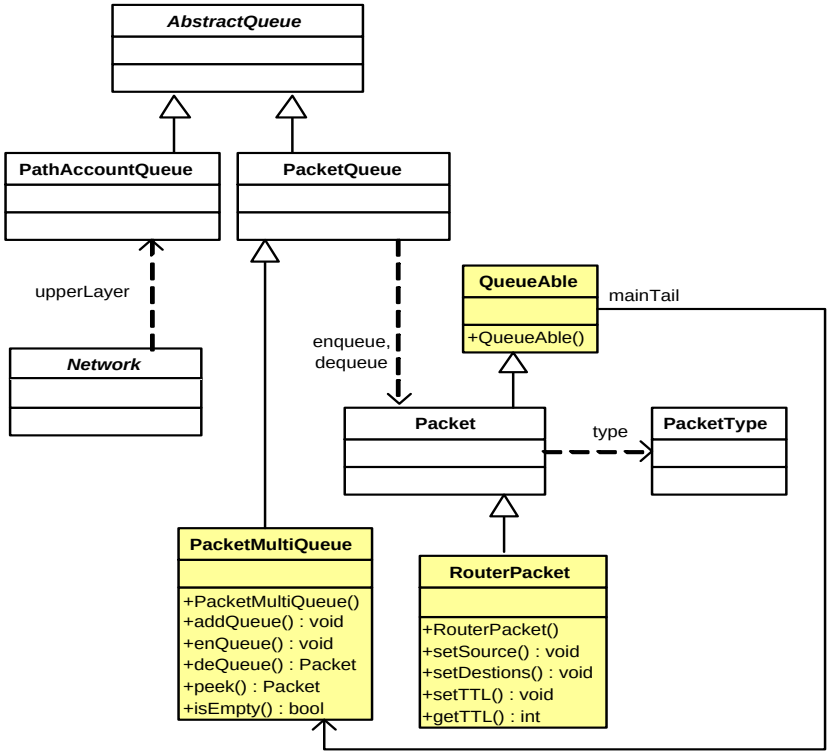


FIGURA 5.3 – Diagrama de Classes do Pacote Network com Suporte a Roteamento

`RouterPacket`

Classe que estende `Packet`, define um tipo de pacote que permite redefinição dos campos que indicam os nodos origem e destino de um pacote. Possui os métodos `setSource()` e `setDestination()`, respectivamente, para isto. Possui ainda os métodos setTTL() e getTTL() para simulações que envolvam controle de *time-to-live,* o qual permite limitar o número de *hops* trafegados por um pacote.

`QueueAble`

Classe que estabelece um pacote estará disponível para acesso tanto em uma fila única para todas as interfaces de entrada (denominada *mainqueue),* como em uma fila específica de determinada interface de entrada do roteador.

`PacketMultiQueue`

Classe que implementa um conjunto de filas de pacotes, as quais também podem ser vistas como uma única fila simples, denominada *mainqueue.* Todas as filas, incluindo *mainqueue*, seguem a política FIFO e tem seu tamanho definido pela aplicação, podendo ser estabelecido como ilimitado. Possui os métodos `addQueue()`, que permite registrar uma nova fila, `enQueue()`, que adiciona um pacote no fim de uma fila, `deQueue()` que retira um pacote do início da fila e retorna como parâmetro de saída do método, `peek()` que permite consultar o próximo elemento e `isEmpty()` que informa se a fila está vazia ou não.

5.3.2 Pacote Node

A Figura 5.4 apresenta o diagrama de classes com suporte a roteamento do pacote Node. A seguir é apresentada a descrição das novas classes do pacote Node, `HostNode` e `RouterNode`.

`HostNode`

Classe abstrata que estende `Node` e contém a lógica da aplicação ou protocolo de transporte a ser simulado. Um nodo que estende `HostNode` contém apenas uma interface de saída para um nodo roteador. Contém os métodos `initialize()`, o qual prepara as estruturas de dados para a execução da simulação e `addPath()` que cria um caminho entre um nodo *host* e outro roteador.

`RouterNode`

Classe abstrata que estende `Node`, é responsável pelo encaminhamento de pacotes na rede. Um nodo que estende `RouterNode` pode ser conectado a um número qualquer de nodos *hosts* e roteadores. A implementação de uma arquitetura de roteamento no SIMMCAST consiste de duas partes: uma estratégia de encaminhamento e um algoritmo de roteamento, o qual decide para que interfaces um pacote recebido deve ser transmitido.

Possui os métodos `encapsulate()`, que cria um novo pacote, dito encapsulado, o qual contém como área de dados o pacote recebido pela interface de entrada, e `setHeaderSize()` que permite a definição do tamanho da área de controle do pacote. Este novo pacote é, então, enviado para a interface de saída do roteador, conforme estratégia de roteamento do roteador. A classe contém, ainda, os métodos `isNeighborHost()`, que indica se determinado nodo *host* está diretamente conectado ao roteador, e `isNeighborRouter()`, que indica se dado nodo roteador é vizinho deste, bem como `isNeighbor()` que também verifica se determinado nodo, identificado como parâmetro de entrada, é vizinho do roteador. Adicionalmente, o método `getNeighborHostsInGroup()` retorna quais *hosts* assinam determinado grupo multicast. Possui ainda os métodos `isNeighborRouterCount()` que indica quantos nodos vizinhos são roteadores, `neighborRouterIds()` que informa os identificadores dos nodos roteadores vizinhos.

5.3.3 Pacote Route

O pacote Route é o novo pacote adicionado ao código do simulador de modo a oferecer a infra-estrutrura necessária na arquitetura a fim de permitir simulações com roteamento. É composto por classes que gerenciam os recursos necessários para o uso de roteamento no SIMMCAST. São elas: `DataPacket`, `UnicastControlPacket`, `MulticastControlPacket`, `GraphEdge`, `DefaultRouterNode`, `DefaultRouterSender`, `DefaultRouterNodeReceiver`, `RoutingAlgorithmStrategy`, `StaticAlgorithmStrategy`, `RoutingTable`, `RoutingTableTuple` e `ShortestPathTree`. As Figuras 5.5 e 5.6 apresentam o diagrama de classes UML do pacote Route. Na seqüência é apresentada a descrição de cada classe.

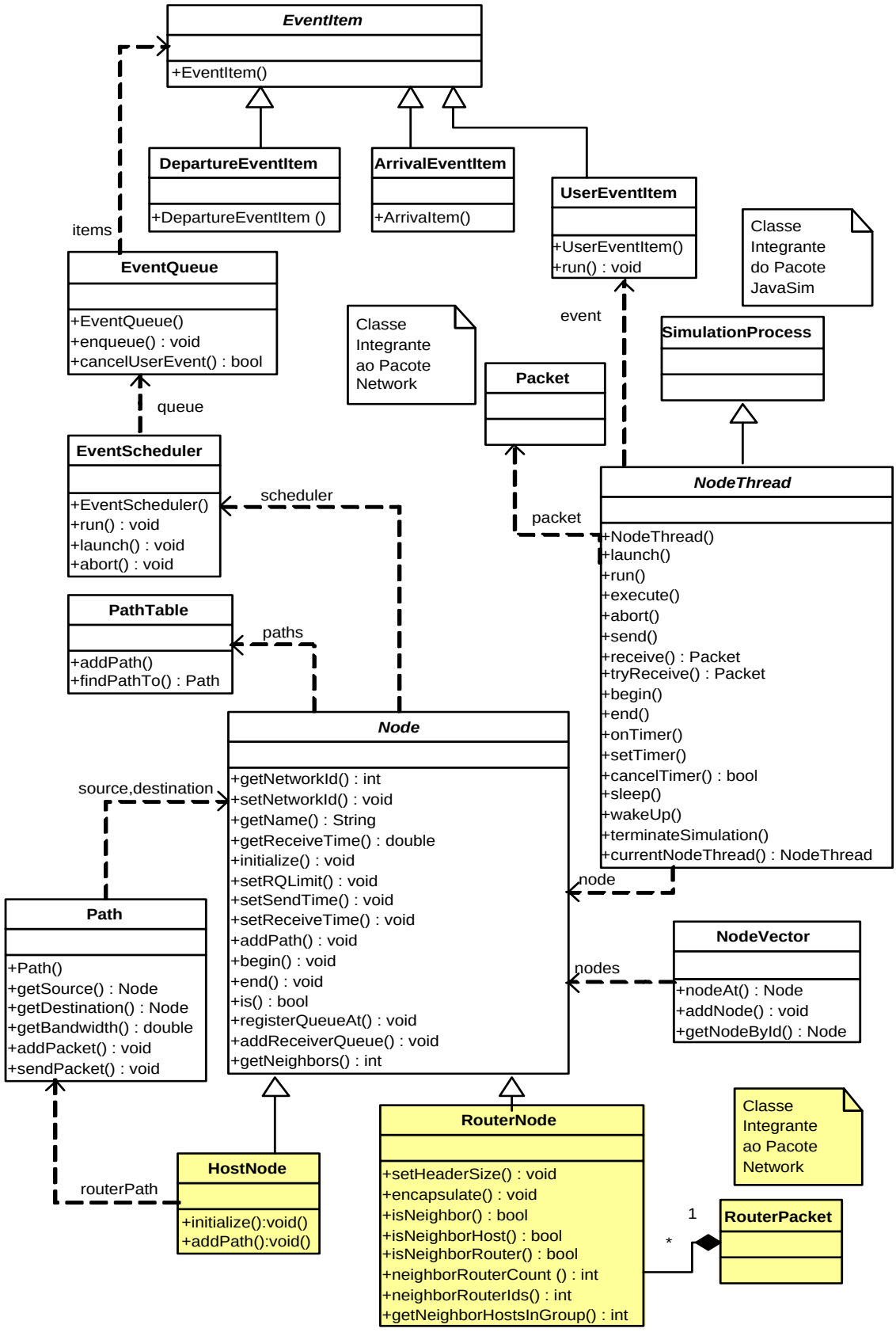


FIGURA 5.4 – Diagrama de Classes do Pacote Node com Suporte a Roteamento

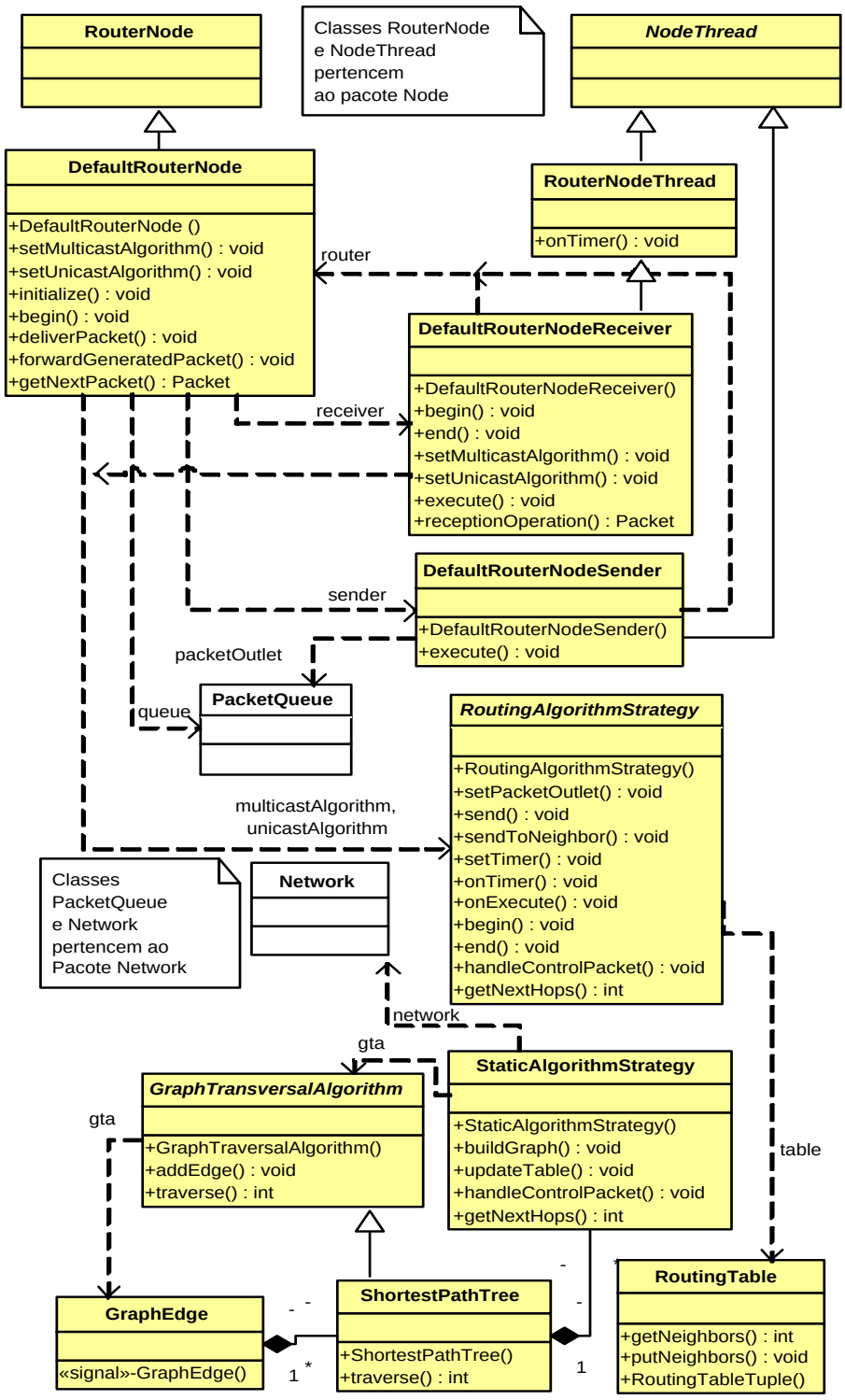


FIGURA 5.5 – Diagrama de Classes do Pacote Route (1)

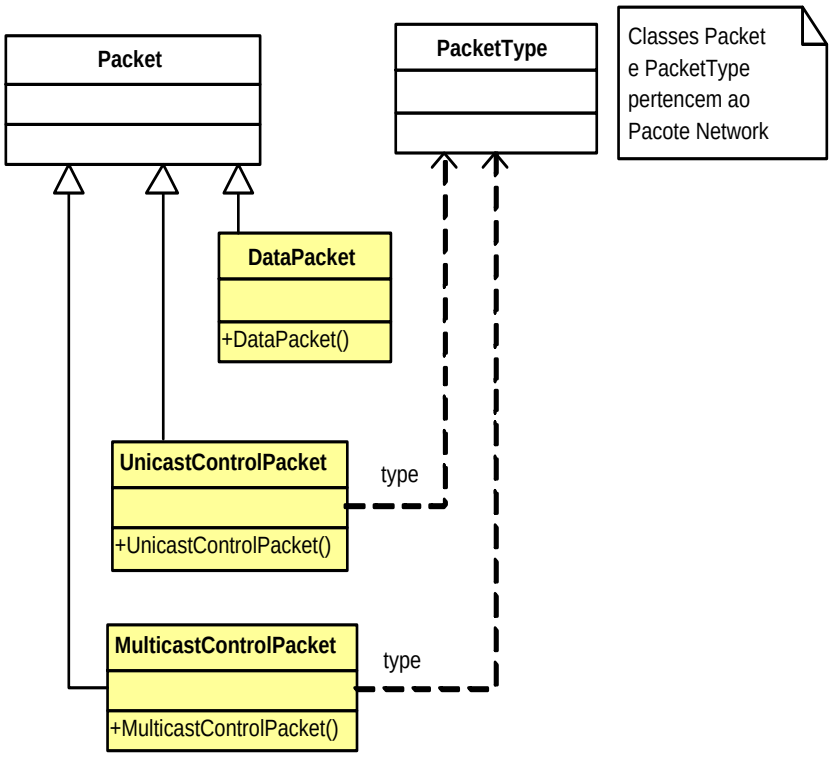


FIGURA 5.6 – Diagrama de Classes do Pacote Route (2)

`DataPacket`

Classe que representa um pacote de dados.

`UnicastControlPacket`

Representa um pacote de controle unicast.

`MulticastControlPacket`

Representa um pacote de controle multicast.

`GraphEdge`

Classe abstrata base da família de algoritmos de processamento de grafos.

`DefaultRouterNode`

Classe que implementa o roteador padrão, que preenche automaticamente a tabela de roteamento existente através da execução de algum algoritmo de busca de rotas, sempre que houver alguma alteração na topologia da rede que está sendo simulada. Não há troca de pacotes na rede e o tempo de convergência é zero. A Figura 5.7 apresenta a arquitetura de um nodo que estende `DefaultRouteNode`. A arquitetura do nodo mostra as interfaces e as filas de entrada e saída, uma para cada *link* existente entre este roteador e outro nodo qualquer, *threads* de recepção e transmissão

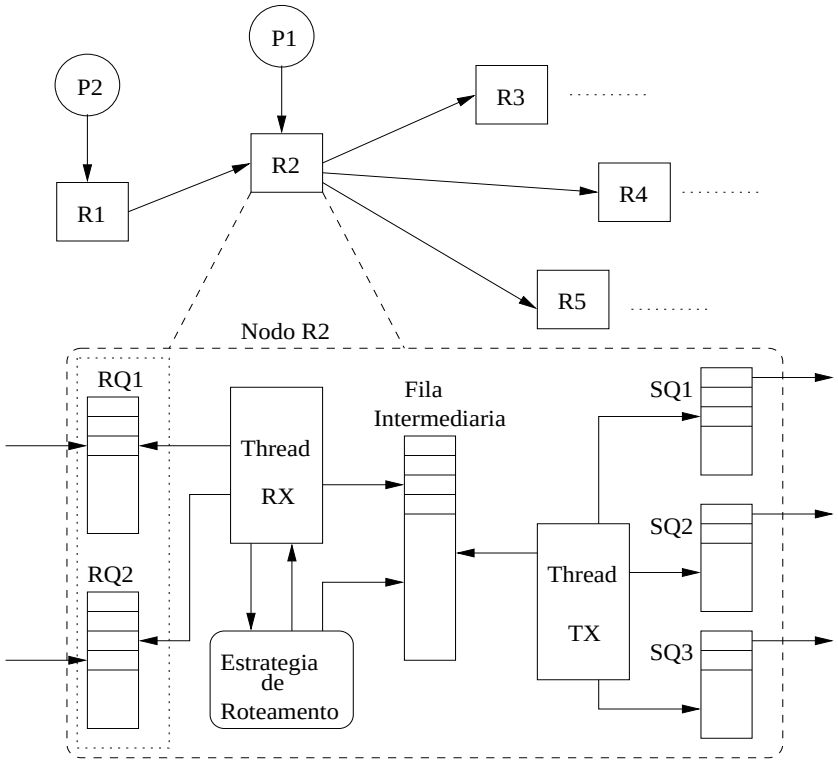


FIGURA 5.7 – Arquitetura de um Nodo Roteador

de pacotes. A *thread* de recepção do nodo é responsável por consumir dados das filas de entrada e consultar a estratégia de roteamento para efetuar o encaminhamento do pacote, que são então colocados em uma fila intermediária, enquanto aguardam que a *thread* de transmissão do nodo os retire e os enviem pela(s) devida(s) interface(s) de saída.

Esta classe possui os seguintes atributos:

- `receiver`: *thread* de recepção de pacotes;
- `sender`: *thread* de transmissão de pacotes;
- `queue`: fila intermediária que armazena pacotes que foram preparados para serem encaminhados. Esta é uma fila interna, de tamanho "infinito". Esta fila não é mostrada no arquivo de traço, uma vez que é utilizada somente como mecanismo de comunicação entre as *threads* produtora (recepção) e consumidora (transmissão) de pacotes;
- `multicastAlgorithm`: representa a estratégia de roteamento multicast;
- `unicastAlgorithm`: representa a estratégia de roteamento unicast.

A classe `DefaultRouterNode` possui os métodos:

- `setMulticastAlgorithm()`: estabelece o algoritmo de roteamento a ser utilizado nas transmissões multicast. Este método foi disponibilizado para ser chamado pelo arquivo de definição da simulação.

- `setUnicastAlgorithm()`: configura o algoritmo de roteamento a ser utilizado nas transmissões unicast. Este método, igualmente, foi disponibilizado para ser chamado pelo arquivo de definição da simulação.
- `initialize()`: método de inicialização do nodo. Define as *threads* de recepção e transmissão de pacotes.
- `begin()`: inicia *threads* de recepção e transmissão de pacotes. Este método é chamado automaticamente pelo objeto `network`.
- `deliverPacket()`: método que entrega pacotes para a *thread* de transmissão. Este método é chamado pela *thread* recepção.
- `getNextPacket()`: consome pacotes da fila de encaminhamento. Se não há pacotes disponíveis, coloca o método de transmissão em *wait* até que algum pacote seja recebido. Este método é chamado pela *thread* de transmissão.
- `forwardGeneratedPacket()`: método responsável pelo encaminhamento de pacotes, verifica o algoritmo de roteamento ativo e executa o encaminhamento de pacotes para outros nodos.

`DefaultRouterNodeSender`

Classe que envia os pacotes preparados (encapsulados ou não) pela *thread* de recepção. Possui o atributo `running`, que indica se a *thread* está executando e o método `execute()` que mantém um *loop* que verifica se novos pacotes foram disponibilizados, e, neste caso, envia o pacote.

`DefaultRouterNodeReceiver`

Thread de recepção do roteador. Contém os atributos: `running` que indica se a *thread* está executando, `multicastAlgorithm` que representa a estratégia de roteamento multicast e `unicastAlgorithm` que representa a estratégia de roteamento unicast.

A classe `DefaultRouterNodeReceiver` possui os métodos `setMulticastAlgorithm()` e `setUnicastAlgorithm()` que associam valores aos atributos correspondentes. Além destes, os método `begin()` e `end()` estão disponíveis para permitir a inclusão de código a ser executado no início e ao final da simulação, respectivamente. Possui ainda o método `receptionOperation()` que permite ao usuário programar sua própria política de recebimento de pacotes nesta *thread*.

`RoutingAlgorithmStrategy`

Classe abstrata que agrega protocolos de roteamento. Esta classe mantém uma tabela de roteamento e suas filas de interface, além de gerenciar os pacotes de controle gerados para a manutenção da tabela de roteamento. Cada nodo roteador deve possuir sua própria instância desta classe. Possui os seguintes atributos:

- `table`: indica a tabela de roteamento;

- `packetOutlet`: fila de saída padrão para o envio de pacotes. Esta fila pode receber pacotes de controle enviados pelos nodos roteadores.

Os seguintes métodos compõem esta classe:

- `setPacketOutlet()`: configura uma fila padrão como interface de saída de pacotes. É chamada pelo nodo roteador antes da estratégia de roteamento ser realmente executada.
- `send()`: método que permite que um pacote seja enviado de um nodo roteador para qualquer outro da rede que está sendo simulada, independente se o nodo é *host* ou roteador.
- `sendToNeighbor()`: este método é uma simplificação do método `send()`, e permite o envio de pacotes somente para nodos diretamente conectados ao roteador.
- `setTimer()`: método que dispara um temporizador.
- `onTimer()`: método que atende interrupção do temporizador.
- `begin()`: método que permite ao usuário adicionar código que executará antes do início da simulação.
- `end()`: método que permite ao usuário adicionar código que será executado no final da simulação.
- `onExecute()`: método que permite ao usuário adicionar código que será executado imediatamente após o início da simulação.
- `handleControlPacket()`: método abstrato que deve ser estendido para o tratamento de pacotes de controle trocados entre os nodos roteadores.
- `getNextHops()`: método abstrato deve ser estendido para conter a lógica do protocolo que determinará os nodos vizinhos para os quais um pacote recebido deve ser enviado.

`StaticAlgorithmStrategy`

Classe que estende a `RoutingAlgorithmStrategy`. Esta classe carrega tanto o protocolo unicast como multicast. Oferece suporte a topologias físicas arbitrárias sem utilizar um protocolo de roteamento. Isto é obtido utilizando um método artificial de atualização da tabela de roteamento, viabilizado pelo conhecimento de mudanças na topologia oferecido pelo pacote Network. Para tal, possui como atributo `network`, que é um identificador para o objeto `network` que contém a composição total da rede, e também `gta`, atributo que contém o grafo de caminhos mais curtos associado. Contém o método `buildGraph()` para construção do grafo e, ainda, a especialização do método abstrato `getNextHops()` da classe `RoutingAlgorithmStrategy`.

`RoutingTable`

`RoutingTable` é a classe que estende `Hashtable`. Representa a tabela de roteamento que será usada pelos objetos `RoutingAlgorithmStrategy`. É uma lista onde as entradas associam uma tupla contendo a origem e o destino do pacote aos nodos vizinhos do nodo. Possui os métodos: `getNeighbors()` que obtém uma entrada da tabela de roteamento, e `putNeighbors()` que adiciona ou atualiza uma entrada na tabela de roteamento.

`RoutingTableTuple`

Tupla que atua como chave da `Hashtable` que implementa a tabela de roteamento.

`ShortestPathTree`

Estende a classe `GraphTraversalAlgorithm`. Possui o método `ShortestPathTree()` que implementa o grafo conforme estratégia de árvore de caminho mais curto de Dijkstra. Para a determinação da árvore os vértices foram configurados como sendo do mesmo tamanho, de modo que os custos das arestas não foram considerados. O método `traverse()` processa o grafo executando o algoritmo.

5.4 Comentários Adicionais

Este capítulo apresentou uma extensão do simulador SIMMCAST para permitir a execução de simulações envolvendo roteamento. Esta nova arquitetura possibilita a execução de simulações sem roteamento ou com roteamento estático.

Já roteamento dinâmico pressupõe que um protocolo de roteamento esteja executando nos nodos roteadores, de modo a garantir a atualização sistemática das informações de roteamento armazenadas. Este capítulo apresentou uma arquitetura capaz de permitir a implementação de protocolos de roteamento multicast e, a partir desta, então, o simulador SIMMCAST passará a suportar experimentos que necessitem roteamento dinâmico.

Capítulo 6

SIMMCAST COM ROTEAMENTO DINÂMICO

O Capítulo 2 apresentou os principais algoritmos de roteamento multicast, enquanto que o Capítulo 3 descreveu brevemente as principais ferramentas para simulação de redes. O Capítulo 4 revisou a estrutura pré-existente do SIMMCAST e ilustra sua utilização. O Capítulo 5 apresentou a extensão do *framework* do SIMMCAST para suportar topologias físicas arbitrárias e roteamento de pacotes através das mesmas; o capítulo ainda descreve uma implementação *default* de roteamento estático. Este suporte permite o desenvolvimento de protocolos estáticos adicionais (por exemplo, baseados em árvores de custo global mínimo) e, particularmente, viabiliza a construção de protocolos de roteamento dinâmico.

O presente capítulo acrescenta ao SIMMCAST (conforme Capítulo 5) dois protocolos dinâmicos para roteamento multicast (vistos no Capítulo 2), um adequado à comunicação com membros de grupos densamente distribuídos (PIM/DM) e outro a grupos com membros esparsos na rede (PIM/SM). Tais protocolos foram escolhidos por possibilitar os estilos de comunicação multicast mais populares, embora não fossem considerados protocolos inter-domínio.

A Seção 6.1 descreve a implementação da versão simulada do PIM/DM no SIMMCAST, e na Seção 6.2 o mesmo para o PIM/SM. Em ambas as seções, primeiro são apresentadas as classes implementadas, seguidas da descrição da implementação e dos principais comandos do protocolo (pacotes de controle). O Anexo A contém o código fonte das referidas implementações.

6.1 Implementação do Protocolo PIM/DM

A Figura 6.1 apresenta o diagrama de classes da implementação do protocolo PIM/DM no SIMMCAST. O código fonte referente encontra-se listado no Anexo A.2. As classes do PIM/DM no SIMMCAST são: `PIMDMNode`, `PIMDMAlgorithmStrategy`, `PIMPacket`, `PIMInterfaceList`, `PIMDMJoinPacket`, `PIMDMPrunePacket`, `PIMDMGraftPacket` e `PIMDM GraftAckPacket`. As mesmas são explicitadas na seqüência:

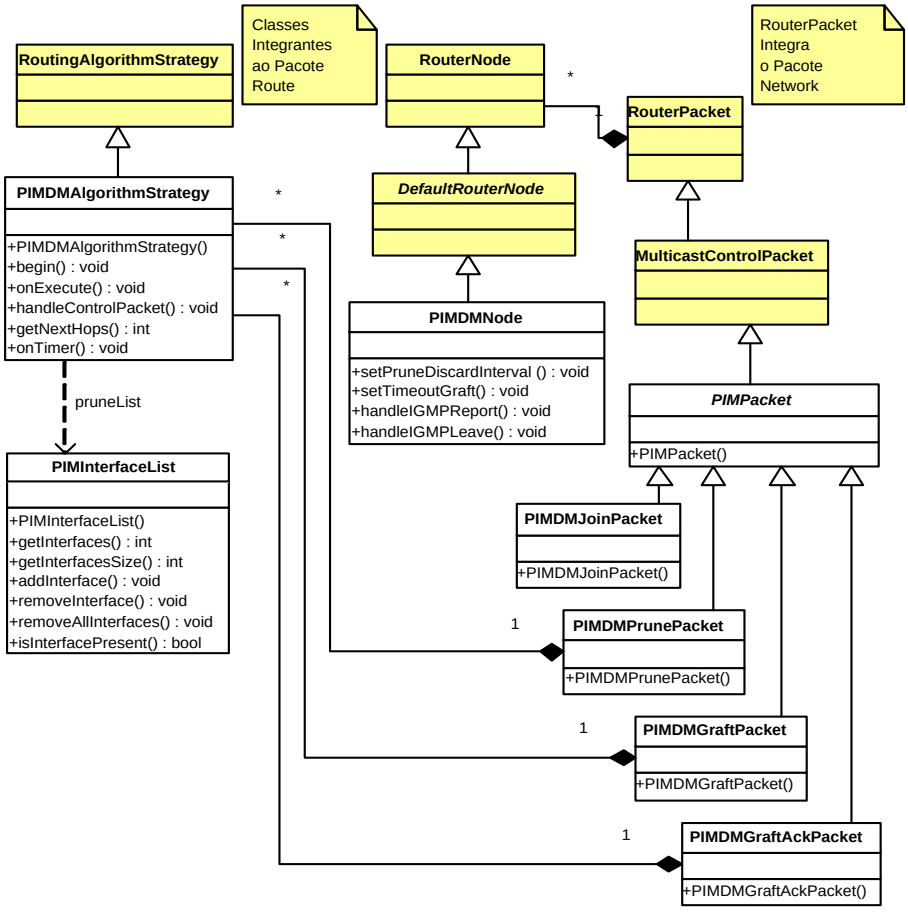


FIGURA 6.1 – Diagrama de Classes do Protocolo PIM/DM

`PIMDMNode`

Classe que estende `DefaultRouterNode`, de modo que herda toda a funcionalidade de recepção e transmissão de pacotes definida para os nodos roteadores já descrita em 5.3.3. Contém os seguintes atributos:

- `pruneList`: lista de que armazena as interfaces que receberam comandos `PIMDMPrunePacket`,bem como informações de origem e grupo associados;
- `pruneDiscardInterval`: atributo que armazena o período em que a lista de *prunes* deve ser descartada, dando início à construção da nova árvore.
- `timeoutGraft`: atributo que armazena o tempo máximo de espera por um comando *graftack* que um roteador que tenha enviado um comando *graft* espera.

A classe `PIMDMNode` possui os métodos `setPruneDiscardInterval()` e `setTimeoutGraft()` que associa os valores lidos do arquivo de definição da simulação aos atributos correspondentes. Além deste possui os métodos `handleIGMPReport()` e `handleIGMPLeave()` que registram *hosts* que assinam determinados grupos multicast.

`PIMDMAlgorithmStrategy`

Classe que estende `RoutingAlgorithmStrategy` e detém a lógica da operação do protocolo. Possui o atributo `staticAlgorithm` que recebe o protocolo unicast em uso na simulação, a partir do arquivo de definição.

Este método é invocado pela *thread* de recepção do nodo responsável por consultar a estratégia de roteamento para efetuar o encaminhamento dos pacotes do roteador para outros nodos da rede. Implementa o método `begin()`, que inicializa o temporizador que controla o descarte da árvore armazenada, bem como o método `onTimer()`, que é executado periodicamente. Ainda, especializa o método `onExecute()` para efetuar o controle de grupos multicast. Possui ainda a especialização do método `handleControlPacket()` que permite tratar a recepção de pacotes de controle do protocolo PIM/DM. Além destes, a especialização do método `getNextHops()` é responsável por determinar os próximos nodos para os quais o pacote recebido será encaminhado. Este método verifica os roteadores vizinhos deste nodo através de chamada ao método `neigborRouterIds()` da classe `RouterNode`.

`PIMPacket`

Classe que estende `MulticastControlPacket`, define um novo pacote conforme tipo e endereços origem e destino especificados.

`PIMInterfaceList`

Classe que implementa uma lista de interfaces. Define métodos para operação desta lista. São `addInterface()`, `removeInterface()` e `removeAllInterfaces()` para inclusão e remoção de interfaces; e `isInterfacePresent()`, `getInterfaces()` e `getInterfacesSize()` para verificação das informações contidas na lista.

`PIMDMJoinPacket`

Classe que estende `PIMPacket`, define um pacote de comando *join*.

`PIMDMPrunePacket`

Classe que estende `PIMPacket`, define um pacote de comando *prune*.

`PIMDMGraftPacket`

Classe que estende `PIMPacket`, define um pacote de comando *graft*.

`PIMDMGraftAckPacket`

Classe que estende `PIMPacket`, define um pacote de comando *graftack*.

Conforme mencionado na Seção 2.2.3, o protocolo PIM/DM é independente de um protocolo de roteamento *unicast* específico. Os roteadores não computam rotas específicas *multicast*. Desta forma, não são estabelecidas tabelas de roteamento multicast nos roteadores, e sim uma lista de informações que armazena os comandos `PIMDMPrunePacket` recebidos, chamada `pruneList,` em cada roteador. Na `pruneList` são armazenadas informações de endereço origem do roteador que enviou o comando, grupo destino e interface de entrada do comando no roteador.

O encaminhamento de pacotes de dados multicast é auxiliado pelo método `getNextHops()` na classe `PIMDMAlgorithmStrategy` de cada roteador, o qual fornece uma lista dos próximos nodos para os quais o pacote recebido será transmitido. Inicialmente são obtidos os endereços origem e destino da mensagem, os quais estão encapsulados na área de dados do pacote recebido. Os seguintes procedimentos são executados na seqüência:

1. De posse das informações de endereço origem e destino da mensagem, o roteador verifica em sua tabela de rotas unicast se a interface de rede pela qual o pacote multicast chegou é a que é usada para enviar pacotes unicast para o computador origem, ou seja, faz uma verificação de RPF.

2. Se a verificação falha, o pacote é descartado e um comando `PIMDMPrunePacket` é enviado pela interface que recebeu o pacote.

3. Se o pacote foi recebido pelo menor caminho entre este roteador e a origem, este pacote poderá ser encaminhado pelas demais interfaces do roteador.

4. Neste caso, se o par (S, G)[1] constar na `pruneList`, um comando `PIMDMPrunePacket` é enviado pela interface RPF do roteador. Caso contrário, o pacote é enviado para todas as interfaces do roteador, obtidas através da chamada do método `neighborRouterIds()` da classe `RouterNode` e que não foram descartadas por comandos *prune* anteriores, exceto a RPF.

5. Se não há tais interfaces, o roteador descarta o pacote e envia um comando `PIMDMPrunePacket` para o roteador vizinho (RPF) que encaminhou o pacote de dados multicast.

[1] (fonte, grupo destino)

O resultado final é uma árvore de caminhos da origem a todos os componentes do grupo.

Como visto, os roteadores precisam memorizar os pacotes `PIMDMPrunePacket` associados a cada par (S, G). Desta forma, de tempos em tempos, conforme configuração, no arquivo de simulação, do parâmetro `PruneDiscardInterval`, estas informações são descartadas, sendo necessário uma nova construção da árvore. O descarte da `pruneList` é feito no método `OnTimer()`. Deste modo, há uma adequação das informações armazenadas à qualquer alteração que tenha ocorrido na topologia da rede ou na composição de grupos, e que não tenha sido atualizada através da troca informações de controle.

Nodos *hosts* são conectados à árvore de distribuição multicast através de *designated routers* (DR), ou roteadores designados, que na vida real são roteadores que interligam uma determinada subrede à árvore de distribuição multicast. Como já mencionado no Capítulo 2, o DR é responsável pela manutenção dos grupos multicast. Normalmente isto é feito por protocolos de manutenção de grupo, como o IGMP (vide Seção 2.2). Como o SIMMCAST não contempla a execução de protocolos de manutenção de grupo, os nodos *hosts* que desejam receber pacotes de determinado grupo mandam solicitações explícitas de *report/leave* para o DR. Esta ações são ativadas de forma artificial, sem troca de pacotes entre os nodos *host* e DR. Sempre que um *host* deseja assinar um grupo multicast, primeiro solicita um *report* para o DR, ou um *leave* para deixar de assinar determinado grupo multicast. Os roteadores podem obter a informação resultante destas ações através de chamadas ao método `getNeighborsHostInGroup()` da classe `RouterNode`.

A interpretação de comandos enviados para o roteador é feita no método `handleControlPacket()`, também da classe `PIMDMAlgorithmStrategy`. Na seqüência são apresentados os comandos *like* IGMP *report* e *leave*, bem como os comandos PIM/DM: *join* , *prune*, *graft* e *graftack*.

6.1.1 Comando *Report*

Comando tratado pelo método `handleIGMPReport()` da classe `PIMDMNode`. Quando uma solicitação de IGMP *report* é percebida pelo DR, o roteador verifica se já é interface para o grupo solicitado. Se não for, remove entrada correspondente na `pruneList` e envia um pacote `PIMDMGraftPacket` para os roteadores vizinhos, para ser reintegrado à árvore de distribuição multicast do grupo.

6.1.2 Comando *Leave*

Comando tratado no método `handleIGMPLeave()` da classe `PIMSMNode`. Quando uma solicitação de IGMP *leave* é percebida pelo DR, o roteador verifica se é interface para o grupo solicitado. Se for, adiciona entrada correspondente à interface na `pruneList` e não encaminha mais pacotes para o *host* que se desligou do grupo. Caso o DR não possua nenhum outro nodo *host* que assina o grupo, então envia um pacote `PIMDMPrunePacket` para os roteadores vizinhos, a fim de que se desligue da árvore de distribuição multicast do grupo.

6.1.3 Comando *Join*

Comandos *join* são enviados sempre que um roteador receber um pacote `PIMDMPrune-Packet` e não puder processar por ser interface RPF para o grupo destino do pacote. O comando `PIMDMJoinPacket` é enviado através da interface de entrada do comando `PIMDMPrune` no roteador.

Quando um pacote `PIMDMJoinPacket` é recebido: (1) o receptor de pacotes percebe que tal comando foi recebido; (2) o roteador aceita o pacote se foi recebido pela interface RPF, e neste caso, extrai informações referentes à origem, destino e interface de entrada do pacote. De posse destas informações, o roteador faz uma busca nos registros já existentes na `pruneList` e, caso seja encontrada uma entrada equivalente (gerada através de um comando `PIMDMPrunePacket` prévio), esta é retirada.

6.1.4 Comando *Prune*

Um comando *prune* é estabelecido através da classe `PIMDMPrunePacket` e é enviado por um nodo roteador PIM/DM sempre que este desejar podar um braço de uma árvore de distribuição multicast. Um pacote `PIMDMPrunePacket` é enviado para desfazer um comando `PIMDMGraftPacket` sempre que membros de um grupo não estão mais presentes em determinado braço da árvore multicast.

Quando um comando `PIMDMPrunePacket` é recebido: (1) o receptor da mensagem percebe que tal comando foi recebido; (2) o roteador aceita o pacote se foi recebido pela interface RPF, e neste caso, armazena as informações referentes à origem, destino e interface de entrada do pacote. De posse destas informações, o roteador faz uma busca nos registros já existentes na `pruneList` e, caso não seja encontrada uma entrada equivalente, esta informação é armazenada.

6.1.5 Comandos *Graft/GraftAck*

São estabelecidos através das classes `PIMDMGraftPacket` e `PIMDMGraftAckPacket`. Um comando `PIMDMGraftPacket` é enviado pelo DR quando um nodo assina determinado grupo e se torna necessário que este roteador se junte à árvore de distribuição multicast do mesmo. Este comando é enviado para todos os roteadores vizinhos do DR.

Quando um comando `PIMDMGraftPacket` é recebido por um roteador, este aceita o pacote se foi recebido pela interface RPF, e neste caso, extrai informações referentes à origem, destino e interface de entrada do pacote. De posse destas informações, o roteador faz uma busca nos registros já existentes na `pruneList` e, caso seja encontrada uma entrada equivalente (gerada através de um comando `PIMDMPrunePacket` prévio), esta é retirada e o pacote encaminhado.

Um roteador que envia um comando `PIMDMGraftPacket` necessita confirmação através do recebimento de um pacote `PIMDMGraftAckPacket`, a fim de que se certifique que continua ligado à árvore de distribuição multicast do grupo. Caso o

pacote de confirmação não seja recebido, o roteador envia periodicamente o comando `PIMDMGraftPacket` até que seja confirmada sua integração à árvore. Isto é feito com auxílio de um vetor de registro de comandos `PIMDMGraftAckPacket` pendentes, `pendingAcks`, de modo que, a cada interrupção do temporizador, é feita uma nova consulta, e caso exista algum comando pendente, um novo comando `PIMDMGraftPacket` é enviado. Comandos `PIMDMGraftAckPacket` são enviados pelos roteadores que recebem comandos `PIMDMGraftPacket`.

6.2 Implementação do Protocolo PIM/SM

A Figura 6.2 apresenta o diagrama de classes da implementação do protocolo PIM/SM no SIMMCAST. O código fonte referente encontra-se listado no Anexo A.3. As classes do PIM/SM no SIMMCAST são: `PIMSMNode`, `PIMSMAlgorithmStrategy`, `PIMPacket`, `PIMSMJoinPacket`, `PIMSMJoinSPTPacket`, `PIMSMJoinRPTPacket`,`PIMSMPrunePacket`, `PIMSMPruneSPTPacket`, `PIMSMPruneRPTPacket`, `PIMSMPruneRPBitPacket`, `PIMSMRegisterPacket`, `PIMSMRegisterStopPacket`,`PIMInterfaceList`, `PIMSMList`, `PIMSMMessagesCountList`,`PIMSMRendezVousPointList` e `PIMSMTimeGroupList`. As mesmas são explicitadas na seqüência:

`PIMSMNode`

Classe que estende `DefaultRouterNode`, de modo que herda toda a funcionalidade de recepção e transmissão de pacotes definida para os nodos. Contém os seguintes atributos:

- `joinList`: lista de que armazena os pacotes *join* recebidos pelos roteadores RP de cada grupo multicast;
- `joinDiscardInterval`: atributo que armazena o período em que a lista de *joins* deve ser descartada;
- `threshold`: atributo que determina o número de mensagens que um nodo receptor receberá através da árvore de distribuição compartilhada, oriundas de um mesmo transmissor. Quando o *threshold* for excedido, o roteador DR conectado ao receptor iniciará o processo para montagem de uma árvore de caminhos mais curtos, SPT, entre o transmissor e o receptor e se desligará da árvore compartilhada, RPT;
- `groupRPList`: atributo que armazena os roteadores núcleo de cada grupo muticast;
- `messageCounterList`: atributo que armazena o número de mensagens recebidas pelo roteador oriundas do fonte S para um mesmo grupo G;
- `kindOfTreeList`: atributo que armazena o tipo de árvore de distribuição do roteador para determinado grupo G;
- `kindOfSourceList`: atributo que armazena informações se, quando operando como fonte, o roteador está utilizando árvore compartilhada ou se está enviando dados para o RP através de comandos *register* para determinado grupo G;

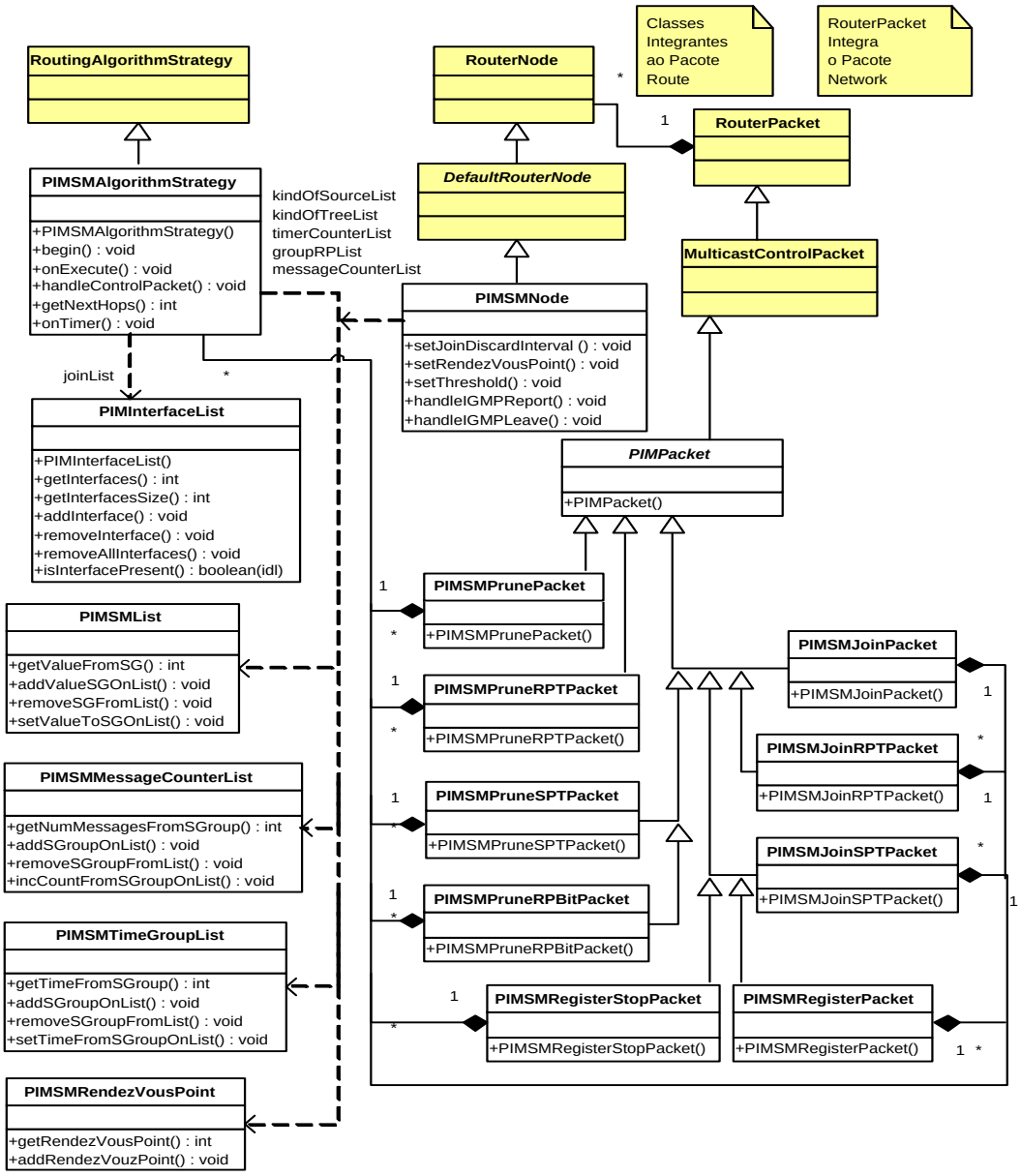


FIGURA 6.2 – Diagrama de Classes do Protocolo PIM/SM

- `timerCountList`: atributo que armazena o tempo de permanência de cada entrada (S,G) ou (*,G) na `joinList`.

A classe `PIMSMNode` possui os métodos `setJoinDiscardInterval()`, `setRendezVousPoint()` e `setThreshold()` que associam os valores lidos do arquivo de definição da simulação aos atributos correspondentes. Da mesma forma que na implementação do protocolo PIM/DM, possui os métodos `handleIGMPReport()` e `handleIGMPLeave()` que registram *hosts* que assinam determinados grupos multicast.

`PIMSMAlgorithmStrategy`

Classe que estende `RoutingAlgorithmStrategy`, detém a lógica da operação do protocolo. Possui o atributo `staticAlgorithm` que recebe o protocolo unicast em uso na simulação, a partir do arquivo de definição da simulação. Implementa o método `begin()` que inicializa o temporizador que controla o descarte das entradas na lista de *joins* armazenada, bem como o método `onTimer()` que é executado periodicamente. Especializa o método `onExecute()` para efetuar o controle de grupos multicast. Possui ainda a especialização do método `handleControlPacket()` que permite controlar a recepção de pacotes de controle do protocolo PIM/SM. Todos os comandos definidos para o protocolo PIM/SM são decodificados neste método. Ainda, a especialização do método `getNextHops()` é reponsável pelo encaminhamento de pacotes de dados para outros nodos, bem como a execução do controle do *threshold* nos roteadores designados dos *hosts* receptores. Este método verifica os roteadores vizinhos deste nodo através de chamada ao método `neigborRouterIds()`.

`PIMPacket`

Classe que estende `MulticastControlPacket`, define um novo pacote conforme tipo e endereços origem e destino especificados.

`PIMSMJoinPacket`

Classe que estende `PIMPacket`, define um pacote de comando *join*.

`PIMSMJoinSPTPacket`

Classe que estende `PIMPacket`, define um pacote de comando *joinSPT*.

`PIMSMJoinRPTPacket`

Classe que estende `PIMPacket`, define um pacote de comando *joinRPT*.

`PIMSMPrunePacket`

Classe que estende `PIMPacket`, define um pacote de comando *prune*.

`PIMSMPruneSPTPacket`

Classe que estende `PIMPacket`, define um pacote de comando *pruneSPT*.

`PIMSMPruneRPTPacket`

Classe que estende `PIMPacket`, define um pacote de comando *pruneRPT*.

`PIMSMPruneRPBitPacket`

Classe que estende `PIMPacket`, define um pacote de comando *pruneRPBit*.

`PIMSMRegisterPacket`

Classe que estende `PIMPacket`, define um pacote de comando *register*. Possui o método `getData()` que devolve o pacote encapsulado na área de dados do pacote `PIMSMRegisterPacket`.

`PIMSMRegisterStopPacket`

Classe que estende `PIMPacket`, define um pacote de comando *registerstop*.

`PIMInterfaceList`

Classe que implementa uma lista de interfaces definindo métodos para operação desta lista: `addInterface()`, `removeInterface()` e `removeAllInterfaces()` para inclusão e remoção de interfaces na lista; e `isInterfacePresent()`, `getInterfaces()` e `getInterfacesSize()` para verificação das informações contidas na lista.

`PIMSMList`

Classe que implementa uma lista de controle genérica. Define os seguintes métodos de acesso à lista: `addSGOnList()`, `removeSGFromList()` para inclusão e remoção de informações na lista, `getValueFromSG()` para consulta de informações e `setValueToSGOnList()` para modificar o valor armazenado na lista.

`PIMSMMessagesCountList`

Classe que define uma lista de controle contendo informações referentes ao número de mensagens recebidas de cada par (S,G) pelo roteador. Estas informações são utilizadas para controle da troca de árvore de SPT para RPT quando o parâmetro *threshold* for excedido. Define os seguintes métodos de acesso à lista: `addSGroupOnList()`, `removeSGroupFromList()` para inclusão e remoção de informações na lista, `getNumMessagesFromSGroup()` para consulta de informações e `incCountFromSGroupOnList()` para incrementar determinado valor armazenado na lista.

`PIMSMRendezVousPointList`

Classe que implementa uma lista contendo informações de quem são os roteadores RP para cada grupo multicast definido. Contém os métodos `addRendezVousPoint()` e `getRendezVousPoint()` para acesso à lista.

`PIMSMTimeGroupList`

Classe que define uma lista de controle contendo informações referentes ao tempo de cada par (S,G) ou (*,G) na `joinList`. Estas informações são utilizadas a cada chamada do temporizador para implementação do *soft-state* pelo roteador. Define os seguintes métodos: `addSGroupOnList()`, `removeSGroupFromList()` para inclusão e remoção de informações na lista, `getTimeFromSGroup()` para consulta de informações e `setTimerFromSGroupOnList()` para modificar o valor armazenado na lista.

O protocolo PIM/SM , como citado na Seção 2.2.3, utiliza um roteador intermediário RP entre a origem e o destino. Os nodos RP de cada grupo multicast são estabelecidos no arquivo de definição da simulação, através do método `setRendezVousPoint()` da classe `PIMSMNode`.

O PIM/SM necessita que os roteadores mantenham estados antes da chegada de pacotes de dados. Para tanto, no encaminhamento de pacotes de dados multicast, os seguintes procedimentos são executados :

1. Cada receptor (*host*) que desejar se associar a um grupo multicast contata seu roteador designado DR, invocando, artificialmente, uma ação de IGMP *report*. Esta ação causa a execução do método `handleIGMPReport()` da classe `PIMSMNode` no DR, o qual dispara um comando `PIMSMJoinPacket` para o RP, se associando para receber pacotes referentes ao grupo especificado.

2. O DR se associa à árvore de distribuição ao enviar uma mensagem explícita de associação para o RP do grupo. Isto é feito através do envio de um comando `PIMSMRegisterPacket` no método `getNextHops()` da classe `PIMSMAlgorithm-Strategy` (vide Figura 6.3).

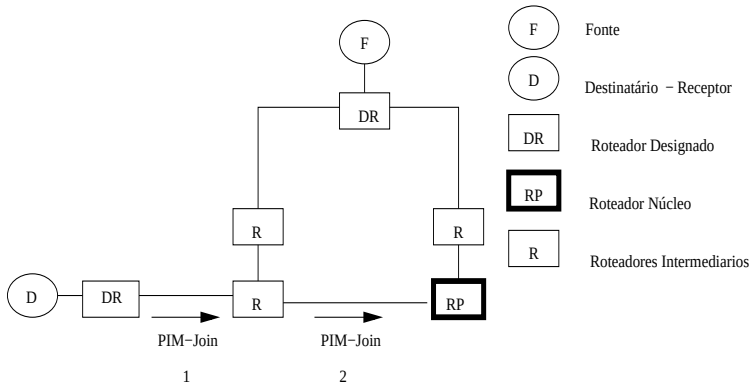


FIGURA 6.3 – Inserção de uma Estação no Grupo Multicast

3. Ao identificar este comando no método `handleControlPacket()` da classe `PIMSMAlgorithmStrategy`, o RP devolve um comando `PIMSMJoinPacket` ao DR fonte, conforme mostra a Figura 6.4.

4. Um nodo fonte utiliza o RP (via DR) para anunciar sua presença e para encontrar um caminho para os membros que tenham se associado ao grupo, conforme mostrado na Figura 6.4.

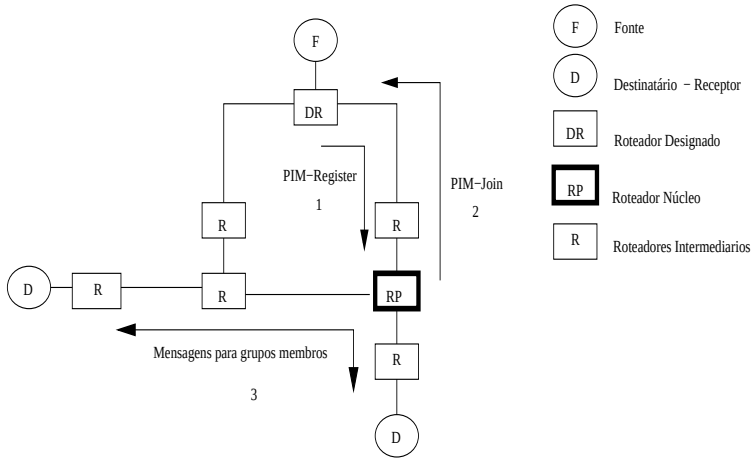


FIGURA 6.4 – Envio de Mensagens Multicast através de RP

5. O DR manda os pacotes multicast encapsulados em pacotes unicast para o RP. Se existem *hosts* receptores que se associaram àquele grupo, o encapsulamento é retirado e o pacote é encaminhado aos receptores. A verificação dos receptores é definido no método `getNextHops()` da classe `PIMSMAlgorithmStrategy`.

6. Se não houver nenhum receptor, o RP manda um comando `PIMSMPrunePacket` para o nodo fonte, a fim de que ele pare de enviar pacotes, economizando banda.

A medida que os roteadores intermediários são informados sobre o RP, transmitem pacotes multicasts encapsulados em pacotes unicasts até o RP. A Figura 6.4 ilustra este procedimento. Quando o tráfego for superior a um limite pré-definido (*threshold*), o roteador DR do fonte S para de encaminhar pacotes ao RP, adotando ao invés, uma árvore de menor caminho baseada no fonte (ou seja, nele mesmo). Desta forma, dependendo da fonte, alguns pacotes podem estar sendo distribuídos através de árvores de distribuição compartilhada (RPT), e outras através de árvores de distribuição baseadas no fonte S (SPT).

A interpretação de comandos enviados para o roteador é feita no método `handleControlPacket()`, também da classe `PIMSMAlgorithmStrategy`. Na seqüência são apresentados os comandos *like* IGMP *report* e *leave*, bem como os comandos PIM/SM: *join, prune, register e registerstop.*

6.2.1 Comando *Report*

Comando tratado no método `handleIGMPReport()` da classe `PIMSMNode`. Quando uma solicitação de IGMP *report* é percebida pelo DR, o roteador verifica se já existe entrada definida para o par (*,G)[2] na `joinList`. Se não está definida: adiciona o par (*,G) na lista, configura que roteador está operando com árvore de distribuição RPT para o par e envia um comando `PIMSMJoinPacket` (*,G) em direção ao RP, consultando a tabela de roteamento unicast para obtenção da interface pela qual deve ser enviado o comando, para que seja reintegrado à árvore de distribuição multicast do grupo G.

[2]onde * significa qualquer fonte

6.2.2 Comando *Leave*

Comando tratado no método `handleIGMPLeave()` da classe `PIMSMNode`. Quando uma solicitação de IGMP *leave* é percebida pelo DR, o roteador verifica se o par (*,G) está presente na `joinList`. Se o par estiver presente, então a entrada (*,G) é removida da lista e é enviado um comando `PIMSMPrunePacket` (*,G) em direção ao RP, também consultando a tabela de roteamento unicast para determinar a interface para a qual será enviado o comando, para que este se desligue da árvore de distribuição multicast do grupo G.

6.2.3 Comando *Join*

Comandos *join* são enviados pelo DR para o RP de determinado grupo sempre que a lista de *host* que enviaram solicitações de IGMP *report* relativos a este grupo deixar de estar vazia porque recebeu um comando `PIMSMJoinPacket` de algum *host* para o grupo especificado.

A implementação do protocolo ainda apresenta alguns comandos específicos de *join*, assim determinados por evidenciarem diferentes procedimentos na interpretação de cada um. São eles:

- *JoinRPT*: *join* enviado pelo RP para um fonte S. Quando o roteador que recebe este comando é o roteador designado DR para o *host* fonte S, configura a árvore de distribuição para RPT. Caso não seja DR, o roteador adiciona o par (S,G) na sua `joinList` e encaminha o comando em direção ao fonte S. Este comando é definido pela classe `PIMSMJoinRPTPacket`.

- *JoinSPT*: *join* enviado pelo roteador para o fonte S quando o número de mensagens recebidas de S por este roteador estoura o *threshold*. Neste caso, se interface (S,G) não está na `joinList`, é adicionada. Se o roteador que recebeu o comando é o roteador designado DR para o *host* fonte S, então troca a árvore de distribuição de RPT para SPT. Este comando é definido pela classe `PIMSMJoinSPTPacket`.

Registros de comandos *join* expiram conforme tempo configurado através do parâmetro `joinDiscardInterval` no arquivo de execução da simulação no SIMMCAST. Desta forma, comandos *join* devem ser enviados periodicamente, tendo em vista as características conceituais do protocolo PIM/SM de encaminhar o tráfego de mensagens somente com solicitação explicita dos *hosts* receptores.

O comportamento periódico do PIM/SM e o refrescamento de estado (*soft-state refresh*) é definido como:

1. A cada minuto é finalizado um temporizador que faz com que o roteador processe a tabela de roteamento inteira (`joinList`) e, para cada entrada:
   se entrada é (*,G), é enviada uma mensagem de *join* (*,G) ao próximo roteador em direção ao RP de G,
   se entrada é (S,G), é enviada uma mensagem de *join* (S,G) ao próximo roteador em direção ao fonte S;

2. Quando um comando *join* (S,G) é recebido, é reinicializado o temporizador da entrada (S,G);

3. Quando um comando *join* (*,G) é recebido, é reinicializado o temporizador da entrada (*,G);

4. Quando é finalizado o temporizador definido em `joinDiscardInterval` de uma entrada, a entrada (S,G) ou (*,G) é removida e um comando *prune* (S,G) ou (*,G) é enviado ao próximo roteador. Caso o parâmetro de configuração `joinDiscardInterval` não tenha sido definido, o valor padrão para descarte é fixado em 3 minutos, conforme definição do protocolo PIM/SM.

6.2.4 Comando *Prune*

Um comando *prune* é enviado por um DR para o RP do grupo sempre que sua lista de *hosts* cadastrados ficar vazia. Neste caso o DR desliga o *host* do RP através do comando `PIMSMPrunePacket`. Quando o RP recebe um comando `PIMSMPrunePacket`, verifica se sua `joinList` ficou vazia para o grupo: se ficou, o roteador envia um comando `PIMSMRegisterStop` para o roteador fonte, para que este pare de enviar pacotes para o RP, uma vez que ninguém mais assina o grupo.

A implementação do protocolo ainda apresenta alguns comandos específicos de *prune*, assim determinados por evidenciarem diferentes procedimentos na interpretação de cada um. São eles:

- *PruneRPT*: *prune* enviado pelo RP para um fonte S. Quando o roteador que recebe este comando é o roteador designado DR para o *host* fonte S, desconecta o roteador da árvore compartilhada RPT. Este comando é definido pela classe `PIMSMPruneRPTPacket`.

- *PruneSPT*: *prune* enviado unicamente pelo RP para um fonte S. Quando um fonte S receber um comando *pruneSPT* remove o par (S,G) da sua `joinList`. Este comando é definido pela classe `PIMSMPruneSPTPacket`.

- *PruneRPBit*: *prune* enviado para o RP pelo roteador DR de um host S. Quando o roteador RP do grupo G recebe este comando, remove a interface (S,G) da sua `joinList`. Este comando é definido pela classe `PIMSMPruneRPBitPacket`.

6.2.5 Comandos *Register/RegisterStop*

Comandos *register* são enviados por um roteador designado para o roteador RP de determinado grupo multicast, quando recebe um pacote de dados de uma fonte S destinado ao grupo G. Estes comandos são comunicações unicasts enviadas pelo DR para o RP, sendo que aquele encapsula o pacote de dados enviado pelo *host* como área de dados do novo pacote de controle representado pelo comando `PIMSMRegisterPacket`. Quando o roteador RP recebe um comando `PIMSMRegisterPacket` verifica se há árvore de distribuição RPT para o grupo G. Se há, desencapsula os dados e encaminha o pacote de dados para os assinantes do grupo G em RP. Além disso, o RP

registra a fonte na `joinList` e envia um comando `PIMSMJoinRPTPacket` *hop-by-hop* até o fonte S.

Comandos `PIMSMRegisterStopPacket` são enviados para os roteadores designados, pelo roteador RP, instruindo-os a pararem de enviar pacotes unicasts em uma das seguintes condições: (1) quando o RP começa a receber tráfego multicast dos nodos fontes ligados ao DR para determinado grupo multicast; (2) quando o RP não possui mais árvore compartilhada para o grupo.

Quando um roteador DR recebe uma mensagem `PIMSMRegisterStopPacket` é terminado o processo de registro e este para de encapsular as mensagens do nodo fonte para o grupo destino especificado através dos pacotes `PIMSMRegisterPacket`.

6.3 Comentários Adicionais

A implementação dos protocolos de roteamento multicast PIM/DM e PIM/SM permitem à ferramenta SIMMCAST executar simulações envolvendo roteamento dinâmico. Por ser o SIMMCAST um *framework*, cabe observar que os mesmos *hot spots* foram estendidos para a implementação dos protocolos. Dada a diferença de especificação e complexidade entre os dois, isto vem ao encontro do que se buscava, que foi, entre outros objetivos, apresentar um modelo para o desenvolvimento de novas implementações.

Como já mencionado, o roteamento no SIMMCAST somente é possível com um mesmo protocolo de roteamento executando em todos os roteadores. Desta forma, é importante esclarecer porque optou-se em não incluir as implementações dos protocolos PIM como integrante do pacote Route do SIMMCAST. Isto foi feito porque decidiu-se deixar o pacote Route somente com a implementação do roteamento estático, o qual é utilizado em todas as simulações envolvendo roteamento, uma vez que este é, também, responsável pelo roteamento unicast na rede.

No entanto, cada protocolo desenvolvido será fornecido como um ente agregado ao simulador, de modo que não sejam necessárias alterações de versão e re-teste do mesmo a cada novo protocolo de roteamento proposto, mas que estarão disponíveis para a utilização de qualquer simulação que venha a utilizar algum protocolo de roteamento dinâmico desde já.

A implementação dos protocolos PIM foi a primeira de uma gama de trabalhos que se inicia a partir desta nova arquitetura do SIMMCAST, abrindo espaço, principalmente, para o estudo de protocolos de roteamento inter-domínios.

Capítulo 7

EXPERIMENTOS

Este capítulo demonstra, através de experimentos dirigidos, a funcionalidade da ferramenta gerada. A Seção 7.1 apresenta a topologia da rede utilizada nas simulações, enquanto a Seção 7.2 demonstra a aplicação desenvolvida para a execução das simulações envolvendo roteamento. Já as Seções 7.3 e 7.4 oferecem avaliações referentes às simulações executadas utilizando roteamento PIM/DM e PIM/SM, respectivamente. A Seção 7.5 tece comentários adicionais referentes aos experimentos realizados.

A arquitetura com roteamento proposta para o SIMMCAST, bem como os protocolos de roteamento desenvolvidos para viabilizar a execução de experimentos de simulação envolvendo roteamento dinâmico, serão aplicados na realização das simulações apresentadas nas próximas sub-seções. Estes experimentos tem como objetivo demonstrar o funcionamento correto da implementação.

7.1 Topologia da Rede

Esta seção apresenta a topologia empregada nas simulações. A topologia utilizada é propositalmente simples, pois busca permitir que se evidencie com clareza a troca de pacotes entre os nodos. A Figura 7.1 ilustra a topologia supra citada, a qual pode ser encontrada em [65], na página 167.

7.2 Descrição da Aplicação

O modelo de aplicação a ser simulado visa a transmissão de pacotes de determinado nodo para um grupo multicast, assinado por um número qualquer de nodos receptores. A montagem da árvore de distribuição multicast, bem como a troca de pacotes entre os roteadores é o objeto de estudo desta simulação. Deste modo, a aplicação utilizada para a execução da simulação tem por objetivo, exclusivamente, demonstrar a funcionalidade dos protocolos de roteamento, não tendo qualquer importância nesta avaliação. Diferentes situações são apresentadas, visando exercitar todas as características de cada protocolo.

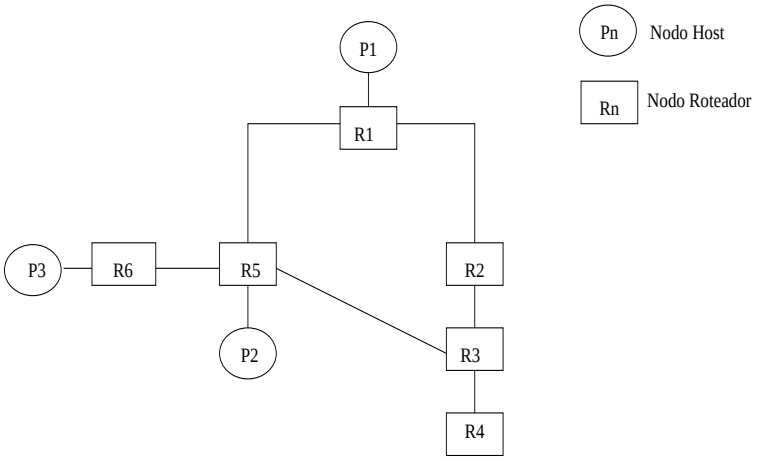


FIGURA 7.1 – Topologia de Rede Empregada nas Simulações

O diagrama UML da aplicação desenvolvida para o simulador é mostrado na Figura 7.2. As classes representadas são: `TestPeer`, `TestSenderThread`, `TestReceiverThread`, `TestNetwork`, `Test`, `TestRouter` e `TestRouterNodeReceiver` e são descritas a seguir.

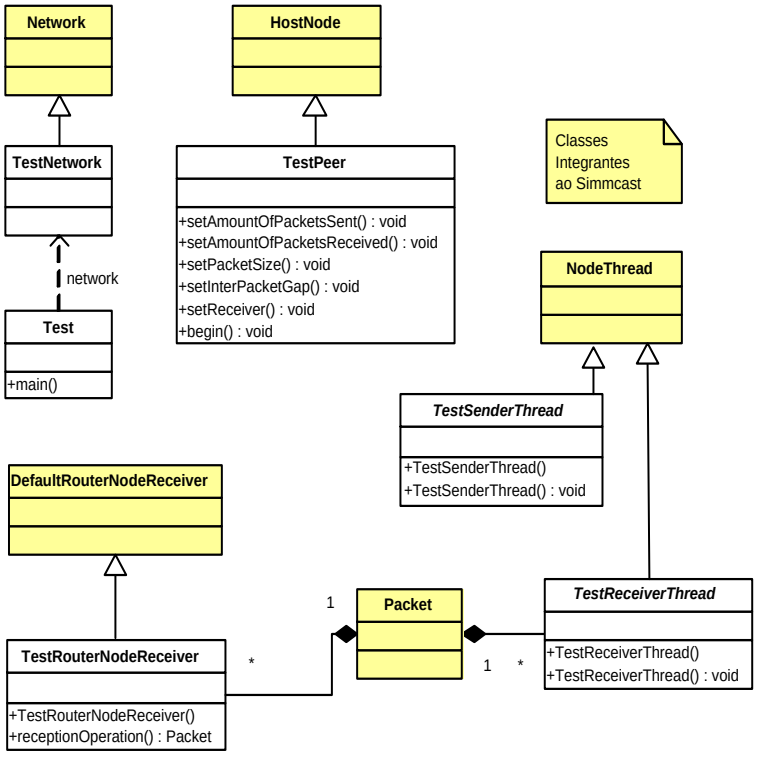


FIGURA 7.2 – Diagrama UML da Aplicação

`TestPeer`

Classe que estende `HostNode` e implementa o comportamento dos nodos *host* da rede. Contém métodos chamados pelo arquivo de configuração e inicia as *threads*

de recepção e transmissão. Contém os métodos `setAmountOfPacketsReceived()`, `setAmountOfPacketsSent()`, `setPacketSize()`, `setInterPacketGap()` e `setReceiver()` que definem os processos de recepção e transmissão na simulação (vide Seção 4.4.1).

`TestSenderThread`

Classe que define comportamento da *thread* de transmissão dos nodos *host* (vide Seção 4.4.2).

`TestReceiverThread`

Classe que define comportamento da *thread* de recepção dos nodos *host* (vide Seção 4.4.2).

`TestNetwork`

Classe que estende `Network`, define a rede para simulação (vide Seção 4.4.3).

`Test`

Classe que implementa a função `main()`, responsável pelo laço de execução da simulação. Verifica se arquivo de definição foi fornecido: se sim, inicia a simulação, caso contrário, solicita arquivo ao usuário (vide Seção 4.4.4).

7.3 Simulação com Protocolo de Roteamento PIM/DM

7.3.1 Configuração da Simulação

A simulação com roteamento utilizando o protocolo PIM/DM, é definida pelo arquivo de configuração *pimdm.sim*, conforme padrão estabelecido na Seção 4.4.5. Para a simulação é definido que o nodo transmissor é o *host* P1, enquanto que os receptores são os *host* P2 e P3, os quais compõem o grupo G1. É estabelecido que P1 enviará 10 pacotes de dados, os quais serão roteados até os receptores P2 e P3 segundo a política de encaminhamento do protocolo denso. Ainda, o valor para descarte das informações de *prune* armazenadas nos roteadores é definido como 15 segundos, através do método `setPruneDiscardInterval()`, e o valor de *timeout* para confirmação de *grafts* como 5 segundos, definido através do método `setTimeoutGraft()`.

É importante verificar que para cada nodo roteador é definida uma estratégia de roteamento unicast e outra multicast, sendo esta última que define que o protocolo em uso é o PIM/DM. Salienta-se aqui que todos os roteadores devem executar o mesmo protocolo de roteamento multicast. Na seqüência é apresentado, comentado, o arquivo de definição das simulações para modo denso.

```
#Define traço da simulação como formato Simmcast
new TRACER simmcast.trace.SimmcastTraceGenerator

#Define arquivo de traco como sendo outdmfile.trace
trace TRACER setFile outdmfile.trace
```

```
#Habilita traço na rede
network setTracer TRACER

#Define nodos host:  TestPeer é classe integrante da aplicação
new P[1..3] TestPeer

#Define nodos roteadores e os define como nodos PIMDM:PIMDMNode é classe integrante da estratégia PIMDM
new R[1..6] simmcast.route.PIMDMNode

#Define tempo de descarte da prunelist:  setPruneDiscardInterval é classe integrante da estratégia PIMDM
R[1..6] setPruneDiscardInterval 15

#Define tempo de timeout de confirmação de graft:  setTimeoutGraft é classe integrante da estratégia PIMDM
R[1..6] setTimeoutGraft 5

#Define estratégia de roteamento unicast e multicast para todos os nodos
#StaticAlgorithmStrategy é classe integrante da estratégia de roteamento estático
#PIMDMAlgorithmStrategy é classe integrante da estratégia de roteamento PIM/DM
new STATIC[n:1..6] simmcast.route.StaticAlgorithmStrategy R[n] network
new PIMDM[n:1..6] simmcast.route.PIMDMAlgorithmStrategy R[n] STATIC[n]
#Atribui estratégias de roteamento ao nodo
#SetMulticastAlgorithm é método integrante do pacote route
R[n:1..6] setMulticastAlgorithm PIMDM[n]
#SetUnicastAlgorithm é método integrante do pacote route
R[n:1..6] setUnicastAlgorithm STATIC[n]

#Define caminhos entre os nodos
#Estabelece caminho entre P1 e R1.  O método addPath é definido na classe HostNode do pacote Node
#São parametros de entrada e seus respectivos valores no exemplo:
#nodo destino = R1, capacidade do link = 100, largura de banda = 1000
#fluxo de propagação = simmcast.stream.FixedStream
#taxa de perda = 0, limite fila recepção = 0 (infinita)
new fixedZero simmcast.stream.FixedStream 0.0
P1 addPath R1 100 1000 fixedZero 0.0
R1 addPath P1 100 1000 fixedZero 0.0
P2 addPath R5 100 1000 fixedZero 0.0
R5 addPath P2 100 1000 fixedZero 0.0
P3 addPath R6 100 1000 fixedZero 0.0
R6 addPath P3 100 1000 fixedZero 0.0
R1 addPath R2 100 1000 fixedZero 0.0
R2 addPath R1 100 1000 fixedZero 0.0
R2 addPath R3 100 1000 fixedZero 0.0
R3 addPath R2 100 1000 fixedZero 0.0
R1 addPath R5 100 1000 fixedZero 0.0
R5 addPath R1 100 1000 fixedZero 0.0
R5 addPath R6 100 1000 fixedZero 0.0
R6 addPath R5 100 1000 fixedZero 0.0
R4 addPath R3 100 1000 fixedZero 0.0
R3 addPath R4 100 1000 fixedZero 0.0
R3 addPath R5 100 1000 fixedZero 0.0
R5 addPath R3 100 1000 fixedZero 0.0

#Define numero de transmissoes do host P1
#setAmountOfPacketsSent é método integrante da aplicação
P1 setAmountOfPacketsSent 10

#Define numero de transmissoes dos demais hosts
P[2..3] setAmountOfPacketsSent 0

#Define numero de recepcoes de cada host
#setAmountOfPacketsReceived é método integrante da aplicação
P1 setAmountOfPacketsReceived 0
P2 setAmountOfPacketsReceived 10
P3 setAmountOfPacketsReceived 10

#Define tempo entre transmissoes
#setInterPacketGap é método integrante da aplicação
P[1..3] setInterPacketGap 2

#Define grupo multicast G1
new G1 simmcast.group.Group
```

```
#Estabelece que o grupo G1 e composto pelos hosts 2 e 3
G1 join P[2..3]

#Define hosts receptores
#setReceiver é método integrante da aplicação e estabelece que P1 enviará para G1
P1 setReceiver G1
```

7.3.2 Avaliação da Execução da Simulação

A partir do arquivo de configuração *pimdm.sim* que define a simulação para o modo denso, a execução da simulação gerou o arquivo de traço *outdmfile.trace*. Dentre as mensagens integrantes do traço, foram selecionadas: (1) as informações relativas à versão do simulador; (2) a correlação entre os nodos e identificadores utilizados; e (3) mensagens relativas à inclusão de pacotes na fila de envio dos nodos roteadores.

No arquivo *outdmfile.trace* pode ser observada a definição da topologia especificada no arquivo de simulação. Além disso, observa-se que o grupo multicast G1 é denotado como endereço destino = -1 (no SIMMCAST os identificadores inteiros maiores que zero identificam nodos, como por exemplo: P1 = 1, de modo que grupos multicast são identificados através de números inteiros negativos) e que cada linha de informação de envio de pacote, apresenta, na seqüência, o tempo em que o pacote foi colocado na fila de transmissão do roteador, o tipo de pacote, um identificador da mensagem gerada, os últimos nodos origem e destino, e, encapsulado como dado, o pacote enviado pelo nodo transmissor, que, como já mencionado, é o *host* P1, nodo 1 da simulação. Através da análise do traço gerado, é possível o acompanhamento do trajeto de cada mensagem enviada pelo transmissor, até que atinjam seus destinatários, os *hosts* P2 e P3, denotados como nodos 2 e 3 no simulador. Os tipos de pacotes definidos pelo protocolo PIM/DM são relacionados na Tabela 7.1, para os quais é feita uma correlação com a classe que os define.

Tipo de Pacote	**Classe Correspondente**	**Referência (página)**
PIMDM_JOIN	PIMDMJoinPacket	73
PIMDM_PRUNE	PIMDMPrunePacket	74
PIMDM_GRAFT	PIMDMGraftPacket	74
PIMDM_GRAFTACK	PIMDMGraftAckPacket	74

TABELA 7.1 – Tipos de Pacotes do Protocolo PIM/DM

O arquivo de traço da simulação, editado para incluir comentários, é apresentado completo a seguir. Na seqüência, partes do traço são comentadas isoladamente para facilitar o entendimento do mesmo, buscando evidenciar o comportamento já apresentado na Seção 6.1 para o protocolo de roteamento PIM/DM.

Para demonstrar a execução do protocolo quatro experimentos foram realizados. O primeiro permite visualizar: (1) o processo de inundação dos pacotes para os vizinhos do roteador; (2) o envio dos comandos `PIMDM_PRUNE` para que seja feita a redução da árvore de distribuição e (3) o descarte das informações armazenadas na `pruneList`,

o que força nova inundação da rede e garante a atualização da árvore. Os demais experimentos demonstram os processos em que: (1) ocorre poda indevida de um ramo da árvore; (2) um nodo *host* que assina o grupo G1 se desliga e posteriormente volta a assiná-lo e (3) ocorre falha no recebimento de confirmação de comandos PIMDM_GRAFT.

```
#versão do framework
SIMMCAST 1.1.5

#criação dos nodos
#para cada nodo definido no arquivo .sim é atribuido um identificador de nodo id
NODE 1 P1
NODE 2 P2
NODE 3 P3
NODE 4 R1
NODE 5 R2
NODE 6 R3
NODE 7 R4
NODE 8 R5
NODE 9 R6

#criação dos links
#estabelece os caminhos bidirecionais entre os nodos
LINK P1 R1
LINK R1 P1
LINK P2 R5
LINK R5 P2
LINK P3 R6
LINK R6 P3
LINK R1 R2
LINK R2 R1
LINK R2 R3
LINK R3 R2
LINK R1 R5
LINK R5 R1
LINK R5 R6
LINK R6 R5
LINK R4 R3
LINK R3 R4
LINK R3 R5
LINK R5 R3

#sintaxe do traço para a primeira informação:
# 0.0 -> relogio da simulacao
# (type ROUTER id 1 from 1 to 4 (data (type APP id 0 from 1 to -1 (data 0)))) -> pacote a ser enviado,
# onde:  id = identificador do pacote, type indica o tipo de pacote em transito,
# como por exemplo:  APP, que sao pacotes enviados pela aplicacao,
# ROUTER, que sao pacotes que encapsulam pacotes APP e os enviam para outro nodo,
# PIMDM_PRUNE. que sao pacotes de controle enviados entre os roteadores, ...
# from x :  nodo origem do pacote
# to y :  nodo destino do pacote
# data (xxx) :  area de dados do pacote.  Pacotes encapsulados por roteadores possuem
# o original na sua area de dados
# SQ -> fila de transmissao que recebe o pacote a ser enviado

0.0 => SQ (type ROUTER id 1 from 1 to 4 (data (type APP id 0 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 5 from 4 to 5 (data (type APP id 4 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 7 from 4 to 8 (data (type APP id 6 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 11 from 8 to 9 (data (type APP id 10 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 13 from 8 to 6 (data (type APP id 12 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 15 from 8 to 2 (data (type APP id 14 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 5 from 5 to 6 (data (type APP id 4 from 1 to -1 (data 0))))
1.8 => SQ (type PIMDM_PRUNE id 20 from 6 to 8 (data null))
1.8 => SQ (type ROUTER id 24 from 6 to 7 (data (type APP id 23 from 1 to -1 (data 0))))
1.8 => SQ (type ROUTER id 11 from 9 to 3 (data (type APP id 10 from 1 to -1 (data 0))))

2.0 => SQ (type ROUTER id 30 from 1 to 4 (data (type APP id 29 from 1 to -1 (data 1))))
2.4 => SQ (type PIMDM_PRUNE id 33 from 7 to 6 (data null))
2.5 => SQ (type PIMDM_PRUNE id 36 from 8 to 6 (data null))
2.6 => SQ (type ROUTER id 40 from 4 to 5 (data (type APP id 39 from 1 to -1 (data 1))))
2.6 => SQ (type ROUTER id 42 from 4 to 8 (data (type APP id 41 from 1 to -1 (data 1))))
```

```
3.2 => SQ (type ROUTER id 46 from 8 to 9 (data (type APP id 45 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 48 from 8 to 2 (data (type APP id 47 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 40 from 5 to 6 (data (type APP id 39 from 1 to -1 (data 1))))
3.8 => SQ (type ROUTER id 46 from 9 to 3 (data (type APP id 45 from 1 to -1 (data 1))))
3.8 => SQ (type PIMDM_PRUNE id 55 from 6 to 5 (data null))

4.0 => SQ (type ROUTER id 57 from 1 to 4 (data (type APP id 56 from 1 to -1 (data 2))))
4.6 => SQ (type ROUTER id 61 from 4 to 5 (data (type APP id 60 from 1 to -1 (data 2))))
4.6 => SQ (type ROUTER id 63 from 4 to 8 (data (type APP id 62 from 1 to -1 (data 2))))
5.1 => SQ (type ROUTER id 67 from 8 to 9 (data (type APP id 66 from 1 to -1 (data 2))))
5.1 => SQ (type ROUTER id 69 from 8 to 2 (data (type APP id 68 from 1 to -1 (data 2))))
5.1 => SQ (type PIMDM_PRUNE id 72 from 5 to 4 (data null))
5.7 => SQ (type ROUTER id 67 from 9 to 3 (data (type APP id 66 from 1 to -1 (data 2))))

6.0 => SQ (type ROUTER id 76 from 1 to 4 (data (type APP id 75 from 1 to -1 (data 3))))
6.6 => SQ (type ROUTER id 76 from 4 to 8 (data (type APP id 75 from 1 to -1 (data 3))))
7.1 => SQ (type ROUTER id 82 from 8 to 9 (data (type APP id 81 from 1 to -1 (data 3))))
7.1 => SQ (type ROUTER id 84 from 8 to 2 (data (type APP id 83 from 1 to -1 (data 3))))
7.7 => SQ (type ROUTER id 82 from 9 to 3 (data (type APP id 81 from 1 to -1 (data 3))))

8.0 => SQ (type ROUTER id 88 from 1 to 4 (data (type APP id 87 from 1 to -1 (data 4))))
8.6 => SQ (type ROUTER id 88 from 4 to 8 (data (type APP id 87 from 1 to -1 (data 4))))
9.2 => SQ (type ROUTER id 94 from 8 to 9 (data (type APP id 93 from 1 to -1 (data 4))))
9.2 => SQ (type ROUTER id 96 from 8 to 2 (data (type APP id 95 from 1 to -1 (data 4))))
9.7 => SQ (type ROUTER id 94 from 9 to 3 (data (type APP id 93 from 1 to -1 (data 4))))

10.0 => SQ (type ROUTER id 100 from 1 to 4 (data (type APP id 99 from 1 to -1 (data 5))))
10.6 => SQ (type ROUTER id 100 from 4 to 8 (data (type APP id 99 from 1 to -1 (data 5))))
11.2 => SQ (type ROUTER id 106 from 8 to 9 (data (type APP id 105 from 1 to -1 (data 5))))
11.2 => SQ (type ROUTER id 108 from 8 to 2 (data (type APP id 107 from 1 to -1 (data 5))))
11.7 => SQ (type ROUTER id 106 from 9 to 3 (data (type APP id 105 from 1 to -1 (data 5))))

12.0 => SQ (type ROUTER id 112 from 1 to 4 (data (type APP id 111 from 1 to -1 (data 6))))
12.6 => SQ (type ROUTER id 112 from 4 to 8 (data (type APP id 111 from 1 to -1 (data 6))))
13.2 => SQ (type ROUTER id 118 from 8 to 9 (data (type APP id 117 from 1 to -1 (data 6))))
13.2 => SQ (type ROUTER id 120 from 8 to 2 (data (type APP id 119 from 1 to -1 (data 6))))
13.7 => SQ (type ROUTER id 118 from 9 to 3 (data (type APP id 117 from 1 to -1 (data 6))))

14.0 => SQ (type ROUTER id 124 from 1 to 4 (data (type APP id 123 from 1 to -1 (data 7))))
14.6 => SQ (type ROUTER id 124 from 4 to 8 (data (type APP id 123 from 1 to -1 (data 7))))
15.2 => SQ (type ROUTER id 130 from 8 to 9 (data (type APP id 129 from 1 to -1 (data 7))))
15.2 => SQ (type ROUTER id 132 from 8 to 6 (data (type APP id 131 from 1 to -1 (data 7))))
15.2 => SQ (type ROUTER id 134 from 8 to 2 (data (type APP id 133 from 1 to -1 (data 7))))
15.7 => SQ (type PIMDM_PRUNE id 137 from 6 to 8 (data null))
15.7 => SQ (type ROUTER id 130 from 9 to 3 (data (type APP id 129 from 1 to -1 (data 7))))

16.0 => SQ (type ROUTER id 141 from 1 to 4 (data (type APP id 140 from 1 to -1 (data 8))))
16.6 => SQ (type ROUTER id 145 from 4 to 5 (data (type APP id 144 from 1 to -1 (data 8))))
16.6 => SQ (type ROUTER id 147 from 4 to 8 (data (type APP id 146 from 1 to -1 (data 8))))
17.2 => SQ (type ROUTER id 151 from 8 to 9 (data (type APP id 150 from 1 to -1 (data 8))))
17.2 => SQ (type ROUTER id 153 from 8 to 2 (data (type APP id 152 from 1 to -1 (data 8))))
17.2 => SQ (type ROUTER id 145 from 5 to 6 (data (type APP id 144 from 1 to -1 (data 8))))
17.8 => SQ (type ROUTER id 151 from 9 to 3 (data (type APP id 150 from 1 to -1 (data 8))))
17.8 => SQ (type ROUTER id 161 from 6 to 7 (data (type APP id 160 from 1 to -1 (data 8))))
17.8 => SQ (type ROUTER id 163 from 6 to 8 (data (type APP id 162 from 1 to -1 (data 8))))

18.0 => SQ (type ROUTER id 165 from 1 to 4 (data (type APP id 164 from 1 to -1 (data 9))))
18.4 => SQ (type PIMDM_PRUNE id 168 from 8 to 6 (data null))
18.4 => SQ (type PIMDM_PRUNE id 171 from 7 to 6 (data null))
18.6 => SQ (type ROUTER id 175 from 4 to 5 (data (type APP id 174 from 1 to -1 (data 9))))
18.6 => SQ (type ROUTER id 177 from 4 to 8 (data (type APP id 176 from 1 to -1 (data 9))))
19.2 => SQ (type ROUTER id 181 from 8 to 9 (data (type APP id 180 from 1 to -1 (data 9))))
19.2 => SQ (type ROUTER id 183 from 8 to 2 (data (type APP id 182 from 1 to -1 (data 9))))
19.2 => SQ (type ROUTER id 175 from 5 to 6 (data (type APP id 174 from 1 to -1 (data 9))))
19.8 => SQ (type ROUTER id 181 from 9 to 3 (data (type APP id 180 from 1 to -1 (data 9))))
19.8 => SQ (type PIMDM_PRUNE id 190 from 6 to 5 (data null))
```

Observando-se as mensagens no arquivo de traço, pode-se ver que, no início da simulação, os pacotes são roteados para todos os vizinhos de cada nodo, exceto para a interface que recebeu o pacote, conforme definição do protocolo PIM/DM. Tomando-se o roteador R1, tem-se que seus vizinhos destinatários são R2 e R5

(nodos 5 e 8 da rede). Quando R1 recebe um pacote enviado por P1, inunda seus vizinhos. Isto pode ser observado no seguinte trecho do traço:

```
0.0 => SQ (type ROUTER id 1 from 1 to 4 (data (type APP id 0 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 5 from 4 to 5 (data (type APP id 4 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 7 from 4 to 8 (data (type APP id 6 from 1 to -1 (data 0))))
```

Da mesma forma, pode-se ver que, quando um roteador percebe que recebeu um pacote por uma interface diferente da RPF, envia um comando `PIMDM_PRUNE` para quem enviou o pacote. O primeiro comando `PIMDM_PRUNE` enviado foi do roteador R3 para R5 (nodos 6 e 8 da rede) , pois o R3 não faz parte da árvore de caminho mais curto entre o nodo transmissor e o roteador R5. Isto pode ser observado na seguinte seqüência do traço:

```
1.2 => SQ (type ROUTER id 13 from 8 to 6 (data (type APP id 12 from 1 to -1 (data 0))))
...[1]
1.8 => SQ (type PIMDM_PRUNE id 20 from 6 to 8 (data null))
```

Outros comandos `PIMDM_PRUNE` são trocados entre os roteadores, de modo que, quando a mensagem que contém no campo de dados o valor 3 for transmitida pelo transmissor, a árvore já convergiu, de modo que não são mais feitas inundações desnecessárias de pacotes. No entanto, devido à características de *soft-state* do protocolo, quando o tempo determinado no arquivo de simulação para que os estados armazenados nos roteadores sejam descartados (`pruneDiscardInterval`), as inundações voltam a ser feitas, de modo a remontar os caminhos de envio dos pacotes do nodo transmissor para os receptores. Pode-se verificar no arquivo de traço que as inundações reiniciam quando da transmissão da mensagem 8, no tempo 16.6, pelo transmissor.

Como no arquivo de definição da simulação foi especificado que a simulação seria finalizada com o envio de 10 mensagens pelo *host* P1, este arquivo de traço representa a totalidade das transações executadas na simulação, sendo esta limitada para facilitar a visualização da execução do protocolo na distribuição dos pacotes multicast entre os nodos da rede.

O experimento apresentado não permite a visualização de características importantes do protocolo. Diante disto, experimentos complementares foram realizados para suprir esta falta, partindo de pequenas diferenças na simulação apresentada.

Inicialmente, tendo-se que, para a topologia utilizada na simulação inicial, o *host* P2 não estivesse assinando o grupo multicast G1, todos os pacotes enviados por P1 seriam descartados por P2 caso P3 não solicitasse a manutenção da árvore através do comando `PIMDM_JOIN` enviado pelo DR de P3 para o DR de P2, imediatamente após receber um comando `PIMDM_PRUNE` para o grupo G1.

A análise do traço gerado mostra que, no tempo 11.2, o roteador R5 desconecta R6 da árvore, e, no tempo 11.8, R6 solicita sua reintegração. A partir daí, os comandos passam a ser repassados até o *host* P3, mesmo que P2 não assine o grupo. Isto é mostrado na seqüência de traço a seguir:

```
0.0 => SQ (type ROUTER id 1 from 1 to 4 (data (type APP id 0 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 5 from 4 to 5 (data (type APP id 4 from 1 to -1 (data 0))))
```

[1] ... significa que existem outras linhas de traço

```
0.6 => SQ (type ROUTER id 7 from 4 to 8 (data (type APP id 6 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 11 from 8 to 9 (data (type APP id 10 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 13 from 8 to 6 (data (type APP id 12 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 15 from 8 to 2 (data (type APP id 14 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 5 from 5 to 6 (data (type APP id 4 from 1 to -1 (data 0))))
1.8 => SQ (type PIMDM_PRUNE id 20 from 6 to 8 (data null))
1.8 => SQ (type ROUTER id 24 from 6 to 7 (data (type APP id 23 from 1 to -1 (data 0))))
1.8 => SQ (type ROUTER id 11 from 9 to 3 (data (type APP id 10 from 1 to -1 (data 0))))

2.0 => SQ (type ROUTER id 30 from 1 to 4 (data (type APP id 29 from 1 to -1 (data 1))))
2.4 => SQ (type PIMDM_PRUNE id 33 from 7 to 6 (data null))
2.5 => SQ (type PIMDM_PRUNE id 38 from 8 to 6 (data null))
2.6 => SQ (type ROUTER id 42 from 4 to 5 (data (type APP id 41 from 1 to -1 (data 1))))
2.6 => SQ (type ROUTER id 44 from 4 to 8 (data (type APP id 43 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 52 from 8 to 9 (data (type APP id 51 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 54 from 8 to 2 (data (type APP id 53 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 42 from 5 to 6 (data (type APP id 41 from 1 to -1 (data 1))))
3.8 => SQ (type ROUTER id 52 from 9 to 3 (data (type APP id 51 from 1 to -1 (data 1))))
3.8 => SQ (type PIMDM_PRUNE id 61 from 6 to 5 (data null))

4.0 => SQ (type ROUTER id 63 from 1 to 4 (data (type APP id 62 from 1 to -1 (data 2))))
4.6 => SQ (type ROUTER id 69 from 4 to 5 (data (type APP id 68 from 1 to -1 (data 2))))
4.6 => SQ (type ROUTER id 71 from 4 to 8 (data (type APP id 70 from 1 to -1 (data 2))))
5.1 => SQ (type ROUTER id 75 from 8 to 9 (data (type APP id 74 from 1 to -1 (data 2))))
5.1 => SQ (type ROUTER id 77 from 8 to 2 (data (type APP id 76 from 1 to -1 (data 2))))
5.1 => SQ (type PIMDM_PRUNE id 80 from 5 to 4 (data null))
5.7 => SQ (type ROUTER id 75 from 9 to 3 (data (type APP id 74 from 1 to -1 (data 2))))

6.0 => SQ (type ROUTER id 86 from 1 to 4 (data (type APP id 85 from 1 to -1 (data 3))))
6.6 => SQ (type ROUTER id 86 from 4 to 8 (data (type APP id 85 from 1 to -1 (data 3))))
7.1 => SQ (type ROUTER id 92 from 8 to 9 (data (type APP id 91 from 1 to -1 (data 3))))
7.1 => SQ (type ROUTER id 94 from 8 to 2 (data (type APP id 93 from 1 to -1 (data 3))))
7.7 => SQ (type ROUTER id 92 from 9 to 3 (data (type APP id 91 from 1 to -1 (data 3))))

8.0 => SQ (type ROUTER id 98 from 1 to 4 (data (type APP id 97 from 1 to -1 (data 4))))
8.6 => SQ (type ROUTER id 98 from 4 to 8 (data (type APP id 97 from 1 to -1 (data 4))))
9.2 => SQ (type ROUTER id 104 from 8 to 9 (data (type APP id 103 from 1 to -1 (data 4))))
9.2 => SQ (type ROUTER id 106 from 8 to 2 (data (type APP id 105 from 1 to -1 (data 4))))
9.7 => SQ (type ROUTER id 104 from 9 to 3 (data (type APP id 103 from 1 to -1 (data 4))))

10.0 => SQ (type ROUTER id 110 from 1 to 4 (data (type APP id 109 from 1 to -1 (data 5))))
10.6 => SQ (type ROUTER id 110 from 4 to 8 (data (type APP id 109 from 1 to -1 (data 5))))
11.2 => SQ (type PIMDM_PRUNE id 115 from 8 to 4 (data null))
11.2 => SQ (type PIMDM_PRUNE id 116 from 8 to 9 (data null))
11.2 => SQ (type PIMDM_PRUNE id 117 from 8 to 6 (data null))
11.8 => SQ (type PIMDM_JOIN id 118 from 9 to 8 (data null))
11.9 => SQ (type ROUTER id 110 from 9 to 3 (data (type APP id 109 from 1 to -1 (data 5))))

12.0 => SQ (type ROUTER id 122 from 1 to 4 (data (type APP id 121 from 1 to -1 (data 6))))
12.6 => SQ (type ROUTER id 122 from 4 to 8 (data (type APP id 121 from 1 to -1 (data 6))))
13.2 => SQ (type ROUTER id 122 from 8 to 9 (data (type APP id 121 from 1 to -1 (data 6))))
13.7 => SQ (type ROUTER id 122 from 9 to 3 (data (type APP id 121 from 1 to -1 (data 6))))

14.0 => SQ (type ROUTER id 130 from 1 to 4 (data (type APP id 129 from 1 to -1 (data 7))))
14.6 => SQ (type ROUTER id 130 from 4 to 8 (data (type APP id 129 from 1 to -1 (data 7))))
15.2 => SQ (type ROUTER id 130 from 8 to 9 (data (type APP id 129 from 1 to -1 (data 7))))
15.7 => SQ (type ROUTER id 130 from 9 to 3 (data (type APP id 129 from 1 to -1 (data 7))))

16.0 => SQ (type ROUTER id 138 from 1 to 4 (data (type APP id 137 from 1 to -1 (data 8))))
16.6 => SQ (type ROUTER id 138 from 4 to 8 (data (type APP id 137 from 1 to -1 (data 8))))
17.2 => SQ (type ROUTER id 138 from 8 to 9 (data (type APP id 137 from 1 to -1 (data 8))))
17.8 => SQ (type ROUTER id 138 from 9 to 3 (data (type APP id 137 from 1 to -1 (data 8))))

18.0 => SQ (type ROUTER id 146 from 1 to 4 (data (type APP id 145 from 1 to -1 (data 9))))
18.6 => SQ (type ROUTER id 146 from 4 to 8 (data (type APP id 145 from 1 to -1 (data 9))))
19.2 => SQ (type ROUTER id 146 from 8 to 9 (data (type APP id 145 from 1 to -1 (data 9))))
19.8 => SQ (type ROUTER id 146 from 9 to 3 (data (type APP id 145 from 1 to -1 (data 9))))
```

Da mesma forma, para a topologia utilizada na simulação inicial, caso o *host* P3 desejasse se desligar do grupo multicast G1, o roteador R6 (DR de P3) envia um comando `PIMDM_PRUNE` para R5, desconectando-se da árvore, uma vez que não possui

mais receptores para o grupo. O envio deste comando é mostrado no tempo 5.7 da simulação. Observe-se que, até então, todos os pacotes enviados por P1 eram recebidos por P2 e P3. Após este tempo, somente P2 passa a receber os pacotes, até que P3 volte a assinar o grupo G1, e consequentemente, R6 se reintegre à árvore. Isto se dá no tempo 20.0 com o envio do comando `PIMDM_GRAFT`. No tempo 20.6 R5 envia o comando de confirmação, `PIMDM_GRAFTACK` para o DR R6. Observe-se que, no tempo 21.8 da simulação, P3 volta a receber os pacotes transmitidos para o grupo. Este processo é mostrado na seqüência de traço a seguir, onde o nodo transmissor P1 envia 12 pacotes para o grupo G1.

```
0.0 => SQ (type ROUTER id 1 from 1 to 4 (data (type APP id 0 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 5 from 4 to 5 (data (type APP id 4 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 7 from 4 to 8 (data (type APP id 6 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 11 from 8 to 9 (data (type APP id 10 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 13 from 8 to 6 (data (type APP id 12 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 15 from 8 to 2 (data (type APP id 14 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 5 from 5 to 6 (data (type APP id 4 from 1 to -1 (data 0))))
1.8 => SQ (type PIMDM_PRUNE id 20 from 6 to 8 (data null))
1.8 => SQ (type ROUTER id 24 from 6 to 7 (data (type APP id 23 from 1 to -1 (data 0))))
1.8 => SQ (type ROUTER id 11 from 9 to 3 (data (type APP id 10 from 1 to -1 (data 0))))

2.0 => SQ (type ROUTER id 30 from 1 to 4 (data (type APP id 29 from 1 to -1 (data 1))))
2.4 => SQ (type PIMDM_PRUNE id 33 from 7 to 6 (data null))
2.5 => SQ (type PIMDM_PRUNE id 38 from 8 to 6 (data null))
2.6 => SQ (type ROUTER id 42 from 4 to 5 (data (type APP id 41 from 1 to -1 (data 1))))
2.6 => SQ (type ROUTER id 44 from 4 to 8 (data (type APP id 43 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 52 from 8 to 9 (data (type APP id 51 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 54 from 8 to 2 (data (type APP id 53 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 42 from 5 to 6 (data (type APP id 41 from 1 to -1 (data 1))))
3.8 => SQ (type ROUTER id 52 from 9 to 3 (data (type APP id 51 from 1 to -1 (data 1))))
3.8 => SQ (type PIMDM_PRUNE id 61 from 6 to 5 (data null))

4.0 => SQ (type ROUTER id 63 from 1 to 4 (data (type APP id 62 from 1 to -1 (data 2))))
4.6 => SQ (type ROUTER id 69 from 4 to 5 (data (type APP id 68 from 1 to -1 (data 2))))
4.6 => SQ (type ROUTER id 71 from 4 to 8 (data (type APP id 70 from 1 to -1 (data 2))))
5.1 => SQ (type ROUTER id 75 from 8 to 9 (data (type APP id 74 from 1 to -1 (data 2))))
5.1 => SQ (type ROUTER id 77 from 8 to 2 (data (type APP id 76 from 1 to -1 (data 2))))
5.1 => SQ (type PIMDM_PRUNE id 80 from 5 to 4 (data null))
5.7 => SQ (type PIMDM_PRUNE id 83 from 9 to 8 (data null))
6.0 => SQ (type ROUTER id 88 from 1 to 4 (data (type APP id 87 from 1 to -1 (data 3))))
6.6 => SQ (type ROUTER id 88 from 4 to 8 (data (type APP id 87 from 1 to -1 (data 3))))
7.1 => SQ (type ROUTER id 88 from 8 to 2 (data (type APP id 87 from 1 to -1 (data 3))))

8.0 => SQ (type ROUTER id 96 from 1 to 4 (data (type APP id 95 from 1 to -1 (data 4))))
8.6 => SQ (type ROUTER id 96 from 4 to 8 (data (type APP id 95 from 1 to -1 (data 4))))
9.2 => SQ (type ROUTER id 96 from 8 to 2 (data (type APP id 95 from 1 to -1 (data 4))))

10.0 => SQ (type ROUTER id 102 from 1 to 4 (data (type APP id 101 from 1 to -1 (data 5))))
10.6 => SQ (type ROUTER id 102 from 4 to 8 (data (type APP id 101 from 1 to -1 (data 5))))
11.2 => SQ (type ROUTER id 102 from 8 to 2 (data (type APP id 101 from 1 to -1 (data 5))))

12.0 => SQ (type ROUTER id 108 from 1 to 4 (data (type APP id 107 from 1 to -1 (data 6))))
12.6 => SQ (type ROUTER id 108 from 4 to 8 (data (type APP id 107 from 1 to -1 (data 6))))
13.2 => SQ (type ROUTER id 108 from 8 to 2 (data (type APP id 107 from 1 to -1 (data 6))))

14.0 => SQ (type ROUTER id 114 from 1 to 4 (data (type APP id 113 from 1 to -1 (data 7))))
14.6 => SQ (type ROUTER id 114 from 4 to 8 (data (type APP id 113 from 1 to -1 (data 7))))
15.2 => SQ (type ROUTER id 114 from 8 to 2 (data (type APP id 113 from 1 to -1 (data 7))))

16.0 => SQ (type ROUTER id 120 from 1 to 4 (data (type APP id 119 from 1 to -1 (data 8))))
16.6 => SQ (type ROUTER id 120 from 4 to 8 (data (type APP id 119 from 1 to -1 (data 8))))
17.2 => SQ (type ROUTER id 120 from 8 to 2 (data (type APP id 119 from 1 to -1 (data 8))))

18.0 => SQ (type ROUTER id 126 from 1 to 4 (data (type APP id 125 from 1 to -1 (data 9))))
18.6 => SQ (type ROUTER id 126 from 4 to 8 (data (type APP id 125 from 1 to -1 (data 9))))
19.2 => SQ (type ROUTER id 126 from 8 to 2 (data (type APP id 125 from 1 to -1 (data 9))))
20.0 => SQ (type PIMDM_GRAFT id 131 from 9 to 8 (data null))

20.0 => SQ (type ROUTER id 133 from 1 to 4 (data (type APP id 132 from 1 to -1 (data 10))))
```

```
20.6 => SQ (type ROUTER id 133 from 4 to 8 (data (type APP id 132 from 1 to -1 (data 10))))
20.6 => SQ (type PIMDM_GRAFTACK id 136 from 8 to 9 (data null))
20.6 => SQ (type PIMDM_GRAFT id 137 from 8 to 6 (data null))
21.2 => SQ (type ROUTER id 141 from 8 to 9 (data (type APP id 140 from 1 to -1 (data 10))))
21.2 => SQ (type ROUTER id 143 from 8 to 2 (data (type APP id 142 from 1 to -1 (data 10))))
21.2 => SQ (type PIMDM_GRAFTACK id 144 from 6 to 8 (data null))
21.2 => SQ (type PIMDM_GRAFT id 145 from 6 to 7 (data null))
21.8 => SQ (type ROUTER id 141 from 9 to 3 (data (type APP id 140 from 1 to -1 (data 10))))
21.8 => SQ (type PIMDM_GRAFTACK id 148 from 7 to 6 (data null))

22.0 => SQ (type ROUTER id 150 from 1 to 4 (data (type APP id 149 from 1 to -1 (data 11))))
22.6 => SQ (type ROUTER id 150 from 4 to 8 (data (type APP id 149 from 1 to -1 (data 11))))
23.2 => SQ (type ROUTER id 156 from 8 to 9 (data (type APP id 155 from 1 to -1 (data 11))))
23.2 => SQ (type ROUTER id 158 from 8 to 2 (data (type APP id 157 from 1 to -1 (data 11))))
23.8 => SQ (type ROUTER id 156 from 9 to 3 (data (type APP id 155 from 1 to -1 (data 11))))

24.0 => SQ (type ROUTER id 162 from 1 to 4 (data (type APP id 161 from 1 to -1 (data 12))))
24.6 => SQ (type ROUTER id 162 from 4 to 8 (data (type APP id 161 from 1 to -1 (data 12))))
25.2 => SQ (type ROUTER id 168 from 8 to 9 (data (type APP id 167 from 1 to -1 (data 12))))
25.2 => SQ (type ROUTER id 170 from 8 to 2 (data (type APP id 169 from 1 to -1 (data 12))))
25.8 => SQ (type ROUTER id 168 from 9 to 3 (data (type APP id 167 from 1 to -1 (data 12))))
```

Outra análise necessária é quando um roteador não recebe confirmação de um comando `PIMDM_GRAFT` enviado. Assim, considerando o experimento anterior, caso R6 não receba confirmação do comando enviado no tempo 15.0, o tempo de espera expira, e um novo comando `PIMDM_GRAFT` é enviado de R6 para R5 no tempo 20.0. Para este comando, é enviado o pacote de confirmação `PIMDM_GRAFTACK` de R5 para R6 no tempo 20.6. Isto pode ser visualizado na seqüência a seguir:

```
0.0 => SQ (type ROUTER id 1 from 1 to 4 (data (type APP id 0 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 5 from 4 to 5 (data (type APP id 4 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 7 from 4 to 8 (data (type APP id 6 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 11 from 8 to 9 (data (type APP id 10 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 13 from 8 to 6 (data (type APP id 12 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 15 from 8 to 2 (data (type APP id 14 from 1 to -1 (data 0))))
1.2 => SQ (type ROUTER id 5 from 5 to 6 (data (type APP id 4 from 1 to -1 (data 0))))
1.8 => SQ (type PIMDM_PRUNE id 20 from 6 to 8 (data null))
1.8 => SQ (type ROUTER id 24 from 6 to 7 (data (type APP id 23 from 1 to -1 (data 0))))
1.8 => SQ (type ROUTER id 11 from 9 to 3 (data (type APP id 10 from 1 to -1 (data 0))))

2.0 => SQ (type ROUTER id 30 from 1 to 4 (data (type APP id 29 from 1 to -1 (data 1))))
2.4 => SQ (type PIMDM_PRUNE id 33 from 7 to 6 (data null))
2.5 => SQ (type PIMDM_PRUNE id 38 from 8 to 6 (data null))
2.6 => SQ (type ROUTER id 42 from 4 to 5 (data (type APP id 41 from 1 to -1 (data 1))))
2.6 => SQ (type ROUTER id 44 from 4 to 8 (data (type APP id 43 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 52 from 8 to 9 (data (type APP id 51 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 54 from 8 to 2 (data (type APP id 53 from 1 to -1 (data 1))))
3.2 => SQ (type ROUTER id 42 from 5 to 6 (data (type APP id 41 from 1 to -1 (data 1))))
3.8 => SQ (type ROUTER id 52 from 9 to 3 (data (type APP id 51 from 1 to -1 (data 1))))
3.8 => SQ (type PIMDM_PRUNE id 61 from 6 to 5 (data null))

4.0 => SQ (type ROUTER id 63 from 1 to 4 (data (type APP id 62 from 1 to -1 (data 2))))
4.6 => SQ (type ROUTER id 69 from 4 to 5 (data (type APP id 68 from 1 to -1 (data 2))))
4.6 => SQ (type ROUTER id 71 from 4 to 8 (data (type APP id 70 from 1 to -1 (data 2))))
5.1 => SQ (type ROUTER id 75 from 8 to 9 (data (type APP id 74 from 1 to -1 (data 2))))
5.1 => SQ (type ROUTER id 77 from 8 to 2 (data (type APP id 76 from 1 to -1 (data 2))))
5.1 => SQ (type PIMDM_PRUNE id 80 from 5 to 4 (data null))
5.7 => SQ (type PIMDM_PRUNE id 83 from 9 to 8 (data null))

6.0 => SQ (type ROUTER id 88 from 1 to 4 (data (type APP id 87 from 1 to -1 (data 3))))
6.6 => SQ (type ROUTER id 88 from 4 to 8 (data (type APP id 87 from 1 to -1 (data 3))))
7.1 => SQ (type ROUTER id 88 from 8 to 2 (data (type APP id 87 from 1 to -1 (data 3))))

8.0 => SQ (type ROUTER id 96 from 1 to 4 (data (type APP id 95 from 1 to -1 (data 4))))
8.6 => SQ (type ROUTER id 96 from 4 to 8 (data (type APP id 95 from 1 to -1 (data 4))))
9.2 => SQ (type ROUTER id 96 from 8 to 2 (data (type APP id 95 from 1 to -1 (data 4))))

10.0 => SQ (type ROUTER id 102 from 1 to 4 (data (type APP id 101 from 1 to -1 (data 5))))
10.6 => SQ (type ROUTER id 102 from 4 to 8 (data (type APP id 101 from 1 to -1 (data 5))))
11.2 => SQ (type ROUTER id 102 from 8 to 2 (data (type APP id 101 from 1 to -1 (data 5))))
```

```
12.0 => SQ (type ROUTER id 108 from 1 to 4 (data (type APP id 107 from 1 to -1 (data 6))))
12.6 => SQ (type ROUTER id 108 from 4 to 8 (data (type APP id 107 from 1 to -1 (data 6))))
13.2 => SQ (type ROUTER id 108 from 8 to 2 (data (type APP id 107 from 1 to -1 (data 6))))

14.0 => SQ (type ROUTER id 114 from 1 to 4 (data (type APP id 113 from 1 to -1 (data 7))))
14.6 => SQ (type ROUTER id 114 from 4 to 8 (data (type APP id 113 from 1 to -1 (data 7))))
15.0 => SQ (type PIMDM_GRAFT id 117 from 9 to 8 (data null))
15.2 => SQ (type ROUTER id 114 from 8 to 2 (data (type APP id 113 from 1 to -1 (data 7))))

16.0 => SQ (type ROUTER id 121 from 1 to 4 (data (type APP id 120 from 1 to -1 (data 8))))
16.6 => SQ (type ROUTER id 121 from 4 to 8 (data (type APP id 120 from 1 to -1 (data 8))))
17.2 => SQ (type ROUTER id 121 from 8 to 2 (data (type APP id 120 from 1 to -1 (data 8))))

18.0 => SQ (type ROUTER id 127 from 1 to 4 (data (type APP id 126 from 1 to -1 (data 9))))
18.6 => SQ (type ROUTER id 127 from 4 to 8 (data (type APP id 126 from 1 to -1 (data 9))))
19.2 => SQ (type ROUTER id 127 from 8 to 2 (data (type APP id 126 from 1 to -1 (data 9))))
20.0 => SQ (type PIMDM_GRAFT id 132 from 9 to 8 (data null))

20.0 => SQ (type ROUTER id 134 from 1 to 4 (data (type APP id 133 from 1 to -1 (data 10))))
20.6 => SQ (type ROUTER id 134 from 4 to 8 (data (type APP id 133 from 1 to -1 (data 10))))
20.6 => SQ (type PIMDM_GRAFTACK id 137 from 8 to 9 (data null))
20.6 => SQ (type PIMDM_GRAFT id 138 from 8 to 6 (data null))
21.2 => SQ (type ROUTER id 142 from 8 to 9 (data (type APP id 141 from 1 to -1 (data 10))))
21.2 => SQ (type ROUTER id 144 from 8 to 2 (data (type APP id 143 from 1 to -1 (data 10))))
21.2 => SQ (type PIMDM_GRAFTACK id 145 from 6 to 8 (data null))
21.2 => SQ (type PIMDM_GRAFT id 146 from 6 to 7 (data null))
21.8 => SQ (type ROUTER id 142 from 9 to 3 (data (type APP id 141 from 1 to -1 (data 10))))
21.8 => SQ (type PIMDM_GRAFTACK id 149 from 7 to 6 (data null))

22.0 => SQ (type ROUTER id 151 from 1 to 4 (data (type APP id 150 from 1 to -1 (data 11))))
22.6 => SQ (type ROUTER id 151 from 4 to 8 (data (type APP id 150 from 1 to -1 (data 11))))
23.2 => SQ (type ROUTER id 157 from 8 to 9 (data (type APP id 156 from 1 to -1 (data 11))))
23.2 => SQ (type ROUTER id 159 from 8 to 2 (data (type APP id 158 from 1 to -1 (data 11))))
23.8 => SQ (type ROUTER id 157 from 9 to 3 (data (type APP id 156 from 1 to -1 (data 11))))

24.0 => SQ (type ROUTER id 163 from 1 to 4 (data (type APP id 162 from 1 to -1 (data 12))))
24.6 => SQ (type ROUTER id 163 from 4 to 8 (data (type APP id 162 from 1 to -1 (data 12))))
25.2 => SQ (type ROUTER id 169 from 8 to 9 (data (type APP id 168 from 1 to -1 (data 12))))
25.2 => SQ (type ROUTER id 171 from 8 to 2 (data (type APP id 170 from 1 to -1 (data 12))))
25.8 => SQ (type ROUTER id 169 from 9 to 3 (data (type APP id 168 from 1 to -1 (data 12))))
```

Estas situações adicionais complementam o experimento relativo ao protocolo PIM/DM.

7.4 Simulação com Protocolo de Roteamento PIM/SM

7.4.1 Configuração da Simulação

A simulação com roteamento utilizando o protocolo PIM/SM, é definida pelo arquivo de configuração *pimsm.sim*, conforme padrão estabelecido na Seção 4.4.5. Da mesma forma que na simulação já apresentada com roteamento PIM/DM, na simulação com roteamento utilizando o protocolo PIM/SM, também é definido o *host* P1 como transmissor, e P2 e P3 como receptores. No entanto, por ser um protocolo mais complexo que PIM/DM, P1 enviará 15 pacotes de dados, para que possam ser evidenciadas mais características do protocolo; entre elas o envio através de comandos `PIMSM_REGISTER`, o envio através da árvore compartilhada e após o chaveamento para as árvores de caminhos mais curtos baseada no fonte. Ainda comandos de *refresh* entre os roteadores poderão ser visualizados. Observa-se também que o nodo RP para o grupo multicast G1 é definido como sendo o roteador R3, através da chamada do método `setRendezVousPoint()`.

É importante verificar que, da mesma forma que para a simulação já apresentada, também é necessário definir para cada nodo roteador uma estratégia de roteamento unicast e outra multicast, sendo esta última define, neste caso, que o protocolo em uso é o PIM/SM. Na seqüência é apresentado, comentado, o arquivo de definição das simulações para modo esparso.

```
#Define traço da simulacao como formato Simmcast
new TRACER simmcast.trace.SimmcastTraceGenerator

#Define arquivo de traco como sendo outsmfile.trace
TRACER setFile outsmfile.trace

#Habilita traço na rede
network setTracer TRACER

#Define nodos host
#TestPeer é classe integrante da aplicação
new P[1..3] TestPeer

#Define nodos roteadores e os define como nodos PIMSM
#PIMSMNode é classe integrante da estratégia PIMSM
new R[1..6] simmcast.route.PIMSMNode

#Define tempo de descarte de entrada na joinlist
#setJoinDiscardInterval é classe integrante da estratégia PIMSM
R[1..6] setJoinDiscardInterval 180

#Define threshold
#setThreshold é classe integrante da estratégia PIMSM
R[1..6] setThreshold 4

#Define estrategia de roteamento unicast e multicast para todos os nodos
#StaticAlgorithmStrategy é classe integrante da estratégia de roteamento estatico
new STATIC[n:1..6] simmcast.route.StaticAlgorithmStrategy R[n] network
#PIMSMAlgorithmStrategy é classe integrante da estratégia de roteamento PIM/SM
new PIMSM[n:1..6] simmcast.route.PIMSMAlgorithmStrategy R[n] STATIC[n]
R[n:1..6] setMulticastAlgorithm PIMSM[n]
R[n:1..6] setUnicastAlgorithm STATIC[n]

#Define caminhos entre os nodos
new fixedZero simmcast.stream.FixedStream 0.0
P1 addPath R1 100 1000 fixedZero 0.0
R1 addPath P1 100 1000 fixedZero 0.0
P2 addPath R5 100 1000 fixedZero 0.0
R5 addPath P2 100 1000 fixedZero 0.0
P3 addPath R6 100 1000 fixedZero 0.0
R6 addPath P3 100 1000 fixedZero 0.0
R1 addPath R2 100 1000 fixedZero 0.0
R2 addPath R1 100 1000 fixedZero 0.0
R2 addPath R3 100 1000 fixedZero 0.0
R3 addPath R2 100 1000 fixedZero 0.0
R1 addPath R5 100 1000 fixedZero 0.0
R5 addPath R1 100 1000 fixedZero 0.0
R5 addPath R6 100 1000 fixedZero 0.0
R6 addPath R5 100 1000 fixedZero 0.0
R4 addPath R3 100 1000 fixedZero 0.0
R3 addPath R4 100 1000 fixedZero 0.0
R3 addPath R5 100 1000 fixedZero 0.0
R5 addPath R3 100 1000 fixedZero 0.0

#Define numero de transmissoes do host1
P1 setAmountOfPacketsSent 15

#Define numero de transmissoes dos demais hosts
P[2..3] setAmountOfPacketsSent 0

#Define numero de recepcoes de cada host
P1 setAmountOfPacketsReceived 0
P2 setAmountOfPacketsReceived 15
P3 setAmountOfPacketsReceived 15
```

```
#Define tempo entre transmissoes
P[1..3] setInterPacketGap 2

#Define grupo multicast G1
new G1 simmcast.group.Group

#Estabelece que o grupo G1 e composto pelos hosts 1, 2 e 3
G1 join P[1..3]

#Define assinatura dos hosts receptores no RP para o grupo G1
#handleIGMPReport é método integrante da estratégia PIMSM
R3 handleIGMPReport G1 P2
R4 handleIGMPReport G1 P3

#Define hosts receptores
P1 setReceiver G1

#Define roteador rendezvous do grupo G1 como sendo R3
#setRendezVousPoint método integrante da estratégia PIMSM
R[1..6] setRendezVousPoint G1 R3
```

7.4.2 Avaliação da Execução da Simulação

A partir do arquivo de configuração *pimsm.sim* que define a simulação para o modo esparso, a execução da simulação gerou o arquivo de traço *outsmfile.trace*. Da mesma forma que para o experimento realizado com roteamento PIM/DM, também foram selecionadas: (1) as informações relativas à versão do simulador; (2) a correlação entre os nodos e identificadores utilizados; e (3) mensagens relativas à inclusão de pacotes na fila de envio dos nodos roteadores, no arquivo de traço.

Na simulação para o modo esparso, o grupo multicast G1 também é denotado como -1, e cada linha de informação de envio de pacote, apresenta, na seqüência, o tempo em que o pacote foi colocado na fila de transmissão do roteador, o tipo de pacote em trânsito, um identificador da mensagem gerada, os últimos nodos origem e destino, e, encapsulado como dado, o pacote enviado pelo nodo transmissor, que nesta simulação também é o *host* P1, nodo 1 na simulação. Os tipos de pacotes definidos pelo protocolo PIM/SM são relacionados na Tabela 7.2, para os quais é feita uma correlação com a classe que os define.

Tipo de Pacote	Classe Correspondente	Referência (página)
PIMSM_PRUNE	PIMSMPrunePacket	79
PIMSM_PRUNERPT	PIMSMPruneRPTPacket	80
PIMSM_PRUNESPT	PIMSMPruneSPTPacket	79
PIMSM_PRUNERPBIT	PIMSMPruneRPBit	80
PIMSM_JOIN	PIMSMJoinPacket	79
PIMSM_JOINRPT	PIMSMJoinRPTPacket	79
PIMSM_JOINSPT	PIMSMJoinSPTPacket	79
PIMSM_REGISTER	PIMSM_Register	80
PIMSM_REGISTERSTOP	PIMSM_RegisterStop	80

TABELA 7.2 – Tipos de Pacotes do Protocolo PIM/SM

O arquivo de traço da simulação, editado para incluir comentários, é apresentado

completo a seguir. No entanto, como a topologia utilizada é a mesma do modo denso, as informações referentes à formação da rede no arquivo de traço da simulação foram omitidas. Na seqüência, partes do traço são comentadas isoladamente para facilitar o entendimento do mesmo, buscando evidenciar o comportamento já apresentado na Seção 6.2 para o protocolo de roteamento PIM/SM.

Para demonstrar a execução do protocolo quatro experimentos foram realizados. O primeiro permite visualizar: (1) o processo em que *hosts* receptores se associam ao RP; (2) o processo de envio de pacotes de dados encapsulados em comandos unicast `PIMSM_REGISTER` para o RP; (3) a distribuição de pacotes através de árvore compartilhada via RP e conseqüente fim do envio mensagens unicast; (4) o chaveamento da árvore compartilhada para uma árvore de caminhos mais curtos entre os *hosts* transmissor e receptor e (5) a manutenção das informações contidas na `joinList` através da troca de mensagens de *refresh* entre os roteadores. Os demais experimentos demonstram os processos em que: (1) um nodo *host* se desliga do grupo, sendo que este experimento permite obsevar a troca de pacotes `PIMSM_PRUNE` e `PIMSM_JOIN` na reconstrução da árvore; (2) nenhum nodo *host* assinou dado grupo multicast junto ao RP e (3) *hosts* receptores assinam o RP mas nenhum pacote é transmitido para o grupo.

```
0.0 => SQ (type PIMSM_JOIN id 2 from 9 to 8 (data null))
0.0 => SQ (type PIMSM_JOIN id 5 from 8 to 6 (data null))
0.0 => SQ (type ROUTER id 7 from 1 to 4 (data (type APP id 6 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 13 from 4 to 5 (data (type PIMSM_REGISTER id 12 from 4 to 6
(data (type APP id 6 from 1 to -1 (data 0))))))
1.2 => SQ (type PIMSM_REGISTER id 12 from 4 to 6 (data (type APP id 6 from 1 to -1 (data 0))))
1.8 => SQ (type ROUTER id 22 from 6 to 8 (data (type APP id 21 from 1 to -1 (data 0))))
1.8 => SQ (type PIMSM_JOINRPT id 25 from 6 to 5 (data null))
2.0 => SQ (type ROUTER id 29 from 1 to 4 (data (type APP id 28 from 1 to -1 (data 1))))
2.4 => SQ (type ROUTER id 35 from 8 to 2 (data (type APP id 34 from 1 to -1 (data 0))))
2.4 => SQ (type ROUTER id 37 from 8 to 9 (data (type APP id 36 from 1 to -1 (data 0))))
2.4 => SQ (type PIMSM_JOINRPT id 46 from 5 to 4 (data null))
2.5 => SQ (type PIMSM_REGISTERSTOP id 26 from 6 to 4 (data null))
2.6 => SQ (type ROUTER id 64 from 4 to 5 (data (type PIMSM_REGISTER id 63 from 4 to 6
(data (type APP id 28 from 1 to -1 (data 1))))))
3.0 => SQ (type ROUTER id 70 from 9 to 3 (data (type APP id 69 from 1 to -1 (data 0))))
3.2 => SQ (type PIMSM_REGISTER id 63 from 4 to 6 (data (type APP id 28 from 1 to -1 (data 1))))
3.8 => SQ (type ROUTER id 111 from 6 to 8 (data (type APP id 110 from 1 to -1 (data 1))))
3.8 => SQ (type PIMSM_JOINRPT id 114 from 6 to 5 (data null))
4.0 => SQ (type ROUTER id 118 from 1 to 4 (data (type APP id 117 from 1 to -1 (data 2))))
4.4 => SQ (type ROUTER id 124 from 8 to 2 (data (type APP id 123 from 1 to -1 (data 1))))
4.4 => SQ (type ROUTER id 126 from 8 to 9 (data (type APP id 125 from 1 to -1 (data 1))))
4.4 => SQ (type PIMSM_JOINRPT id 135 from 5 to 4 (data null))
4.6 => SQ (type ROUTER id 118 from 4 to 5 (data (type APP id 117 from 1 to -1 (data 2))))
5.0 => SQ (type PIMSM_JOINSPT id 156 from 9 to 8 (data null))
5.0 => SQ (type ROUTER id 158 from 9 to 3 (data (type APP id 157 from 1 to -1 (data 1))))
5.6 => SQ (type PIMSM_JOINSPT id 201 from 8 to 4 (data null))

6.0 => SQ (type ROUTER id 203 from 1 to 4 (data (type APP id 202 from 1 to -1 (data 3))))
6.2 => SQ (type PIMSM_PRUNERPBIT id 208 from 4 to 5 (data null))
6.6 => SQ (type ROUTER id 214 from 4 to 8 (data (type APP id 213 from 1 to -1 (data 3))))
6.8 => SQ (type PIMSM_PRUNERPBIT id 221 from 5 to 6 (data null))
7.1 => SQ (type PIMSM_JOINSPT id 226 from 8 to 4 (data null))
7.1 => SQ (type ROUTER id 228 from 8 to 2 (data (type APP id 227 from 1 to -1 (data 3))))
7.1 => SQ (type ROUTER id 230 from 8 to 9 (data (type APP id 229 from 1 to -1 (data 3))))
7.4 => SQ (type PIMSM_PRUNERPT id 249 from 6 to 5 (data null))
7.7 => SQ (type ROUTER id 255 from 9 to 3 (data (type APP id 254 from 1 to -1 (data 3))))

7.8 => SQ (type PIMSM_PRUNERPBIT id 262 from 4 to 5 (data null))
8.0 => SQ (type ROUTER id 280 from 1 to 4 (data (type APP id 279 from 1 to -1 (data 4))))
8.0 => SQ (type PIMSM_PRUNERPT id 285 from 5 to 4 (data null))
8.4 => SQ (type PIMSM_PRUNERPBIT id 298 from 5 to 6 (data null))
8.6 => SQ (type ROUTER id 304 from 4 to 8 (data (type APP id 303 from 1 to -1 (data 4))))
9.0 => SQ (type PIMSM_PRUNERPT id 313 from 6 to 5 (data null))
```

```
9.2 => SQ (type ROUTER id 319 from 8 to 2 (data (type APP id 318 from 1 to -1 (data 4))))
9.2 => SQ (type ROUTER id 321 from 8 to 9 (data (type APP id 320 from 1 to -1 (data 4))))
9.6 => SQ (type PIMSM_PRUNERPT id 342 from 5 to 4 (data null))
9.7 => SQ (type ROUTER id 348 from 9 to 3 (data (type APP id 347 from 1 to -1 (data 4))))

10.0 => SQ (type ROUTER id 368 from 1 to 4 (data (type APP id 367 from 1 to -1 (data 5))))
10.6 => SQ (type ROUTER id 386 from 4 to 8 (data (type APP id 385 from 1 to -1 (data 5))))
11.2 => SQ (type ROUTER id 394 from 8 to 2 (data (type APP id 393 from 1 to -1 (data 5))))
11.2 => SQ (type ROUTER id 396 from 8 to 9 (data (type APP id 395 from 1 to -1 (data 5))))
11.7 => SQ (type ROUTER id 418 from 9 to 3 (data (type APP id 417 from 1 to -1 (data 5))))
12.0 => SQ (type ROUTER id 438 from 1 to 4 (data (type APP id 437 from 1 to -1 (data 6))))
12.6 => SQ (type ROUTER id 452 from 4 to 8 (data (type APP id 451 from 1 to -1 (data 6))))

13.0 => SQ (type PIMSM_JOIN id 457 from 8 to 4 (data null))
13.0 => SQ (type PIMSM_JOIN id 460 from 8 to 6 (data null))
13.0 => SQ (type PIMSM_JOIN id 463 from 9 to 8 (data null))
13.0 => SQ (type PIMSM_JOIN id 466 from 4 to 1 (data null))

13.2 => SQ (type ROUTER id 472 from 8 to 2 (data (type APP id 471 from 1 to -1 (data 6))))
13.2 => SQ (type ROUTER id 474 from 8 to 9 (data (type APP id 473 from 1 to -1 (data 6))))
13.7 => SQ (type ROUTER id 502 from 9 to 3 (data (type APP id 501 from 1 to -1 (data 6))))
14.0 => SQ (type ROUTER id 522 from 1 to 4 (data (type APP id 521 from 1 to -1 (data 7))))
14.6 => SQ (type ROUTER id 536 from 4 to 8 (data (type APP id 535 from 1 to -1 (data 7))))
15.2 => SQ (type ROUTER id 544 from 8 to 2 (data (type APP id 543 from 1 to -1 (data 7))))
15.2 => SQ (type ROUTER id 546 from 8 to 9 (data (type APP id 545 from 1 to -1 (data 7))))
15.7 => SQ (type ROUTER id 568 from 9 to 3 (data (type APP id 567 from 1 to -1 (data 7))))
16.0 => SQ (type ROUTER id 588 from 1 to 4 (data (type APP id 587 from 1 to -1 (data 8))))
16.6 => SQ (type ROUTER id 602 from 4 to 8 (data (type APP id 601 from 1 to -1 (data 8))))
17.2 => SQ (type ROUTER id 610 from 8 to 2 (data (type APP id 609 from 1 to -1 (data 8))))
17.2 => SQ (type ROUTER id 612 from 8 to 9 (data (type APP id 611 from 1 to -1 (data 8))))
17.8 => SQ (type ROUTER id 634 from 9 to 3 (data (type APP id 633 from 1 to -1 (data 8))))
18.0 => SQ (type ROUTER id 646 from 1 to 4 (data (type APP id 645 from 1 to -1 (data 9))))
18.6 => SQ (type ROUTER id 668 from 4 to 8 (data (type APP id 667 from 1 to -1 (data 9))))
19.2 => SQ (type ROUTER id 676 from 8 to 2 (data (type APP id 675 from 1 to -1 (data 9))))
19.2 => SQ (type ROUTER id 678 from 8 to 9 (data (type APP id 677 from 1 to -1 (data 9))))
19.8 => SQ (type ROUTER id 700 from 9 to 3 (data (type APP id 699 from 1 to -1 (data 9))))
20.0 => SQ (type ROUTER id 712 from 1 to 4 (data (type APP id 711 from 1 to -1 (data 10))))
20.6 => SQ (type ROUTER id 734 from 4 to 8 (data (type APP id 733 from 1 to -1 (data 10))))
21.2 => SQ (type ROUTER id 742 from 8 to 2 (data (type APP id 741 from 1 to -1 (data 10))))
21.2 => SQ (type ROUTER id 744 from 8 to 9 (data (type APP id 743 from 1 to -1 (data 10))))
21.8 => SQ (type ROUTER id 766 from 9 to 3 (data (type APP id 765 from 1 to -1 (data 10))))
22.0 => SQ (type ROUTER id 778 from 1 to 4 (data (type APP id 777 from 1 to -1 (data 11))))
22.6 => SQ (type ROUTER id 800 from 4 to 8 (data (type APP id 799 from 1 to -1 (data 11))))
23.2 => SQ (type ROUTER id 808 from 8 to 2 (data (type APP id 807 from 1 to -1 (data 11))))
23.2 => SQ (type ROUTER id 810 from 8 to 9 (data (type APP id 809 from 1 to -1 (data 11))))
23.8 => SQ (type ROUTER id 832 from 9 to 3 (data (type APP id 831 from 1 to -1 (data 11))))
24.0 => SQ (type ROUTER id 844 from 1 to 4 (data (type APP id 843 from 1 to -1 (data 12))))
24.6 => SQ (type ROUTER id 866 from 4 to 8 (data (type APP id 865 from 1 to -1 (data 12))))
25.2 => SQ (type ROUTER id 874 from 8 to 2 (data (type APP id 873 from 1 to -1 (data 12))))
25.2 => SQ (type ROUTER id 876 from 8 to 9 (data (type APP id 875 from 1 to -1 (data 12))))
25.8 => SQ (type ROUTER id 898 from 9 to 3 (data (type APP id 897 from 1 to -1 (data 12))))
26.0 => SQ (type ROUTER id 910 from 1 to 4 (data (type APP id 909 from 1 to -1 (data 13))))

26.0 => SQ (type PIMSM_JOIN id 913 from 4 to 1 (data null))
26.0 => SQ (type PIMSM_JOIN id 916 from 8 to 4 (data null))
26.0 => SQ (type PIMSM_JOIN id 919 from 8 to 6 (data null))
26.0 => SQ (type PIMSM_JOIN id 922 from 9 to 8 (data null))

26.6 => SQ (type ROUTER id 944 from 4 to 8 (data (type APP id 943 from 1 to -1 (data 13))))
27.2 => SQ (type ROUTER id 958 from 8 to 2 (data (type APP id 957 from 1 to -1 (data 13))))
27.2 => SQ (type ROUTER id 960 from 8 to 9 (data (type APP id 959 from 1 to -1 (data 13))))
27.8 => SQ (type ROUTER id 982 from 9 to 3 (data (type APP id 981 from 1 to -1 (data 13))))
28.0 => SQ (type ROUTER id 994 from 1 to 4 (data (type APP id 993 from 1 to -1 (data 14))))
28.6 => SQ (type ROUTER id 1016 from 4 to 8 (data (type APP id 1015 from 1 to -1 (data 14))))
29.2 => SQ (type ROUTER id 1024 from 8 to 2 (data (type APP id 1023 from 1 to -1 (data 14))))
29.2 => SQ (type ROUTER id 1026 from 8 to 9 (data (type APP id 1025 from 1 to -1 (data 14))))
29.8 => SQ (type ROUTER id 1048 from 9 to 3 (data (type APP id 1047 from 1 to -1 (data 14))))
```

Observando-se as mensagens no arquivo de traço, pode-se ver que, no início da simulação, comandos `PIMSM_JOIN` são enviadas para o roteador RP, R3, ao qual corresponde a identificação de nodo 6, como resultado de comandos de *report* dos

receptores P2 e P3 (nodos 2 e 3), definidos no arquivo de simulação, para seus roteadores designados, roteadores R5 e R6, nodos 8 e 9, respectivamente. Com isto estes receptores são cadastrados no RP como receptores para o grupo G1. Isto pode ser visto na seguinte seqüência do traço:

```
0.0 => SQ (type PIMSM_JOIN id 2 from 9 to 8 (data null))
0.0 => SQ (type PIMSM_JOIN id 5 from 8 to 6 (data null))
```

Quando o *host* fonte, P1, inicia a transmissão de pacotes, o roteador R1, que é o *first-hop*, envia uma mensagem unicast de registro para o roteador R3 (RP nodo 6), conforme mostra a mensagem no tempo 1.2, o qual encaminha via RPT para o roteador R5 (nodo 8) e que, por sua vez, encaminha para o roteador R6 (nodo 9), sendo que os roteadores encaminham para seus *hosts* receptores diretamente conectados, P2 e P3. A seqüência de traço a seguir exibe o processo:

```
0.0 => SQ (type ROUTER id 7 from 1 to 4 (data (type APP id 6 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 13 from 4 to 5 (data (type PIMSM_REGISTER id 12 from 4 to 6
(data (type APP id 6 from 1 to -1 (data 0))))))
1.2 => SQ (type PIMSM_REGISTER id 12 from 4 to 6 (data (type APP id 6 from 1 to -1 (data 0))))
1.8 => SQ (type ROUTER id 22 from 6 to 8 (data (type APP id 21 from 1 to -1 (data 0))))
...
2.4 => SQ (type ROUTER id 35 from 8 to 2 (data (type APP id 34 from 1 to -1 (data 0))))
2.4 => SQ (type ROUTER id 37 from 8 to 9 (data (type APP id 36 from 1 to -1 (data 0))))
(data(type APP id 28 from 1 to -1 (data 1))))))
3.0 => SQ (type ROUTER id 70 from 9 to 3 (data (type APP id 69 from 1 to -1 (data 0))))
```

O arquivo de traço também permite observar os comandos `PIMSM_JOIN`, tempos 1.8 e 2.4, que trafegam *hop-by-hop* a partir do RP R3 (nodo 6) em direção ao nodo transmissor P1, a fim de gerar em cada roteador as informações necessárias para o envio de mensagens através de árvore multicast compartilhada. Isto pode ser visto no traço abaixo:

```
1.8 => SQ (type PIMSM_JOINRPT id 25 from 6 to 5 (data null))
...
2.4 => SQ (type PIMSM_JOINRPT id 46 from 5 to 4 (data null))
```

Uma vez montada a árvore entre o RP e o *host* fonte, um comando `PIMSM_REGISTER-STOP` é enviado ao roteador R1 (nodo 4), conforme mostra o comando enviado no tempo 2.5 no traço que segue:

```
2.5 => SQ (type PIMSM_REGISTERSTOP id 26 from 6 to 4 (data null))
```

As mensagens nos tempos 5.0 e 5.6 mostram o chaveamento da árvore compartilhada via roteador RP para uma árvore de caminhos mais curtos entre P1 e P2/P3, , uma vez que o *threshold* configurado foi atingido. Isto se dá com o envio de comandos `PIMSM_JOINSPT` para o roteador R1. Quando é feito o chaveamento, um comando `PIMSM_PRUNERPBIT` é enviado ao RP para desconectar o fonte da árvore compartilhada, conforme mostra a mensagem no tempo 6.2. Isto pode ser visto no traço a seguir:

```
5.0 => SQ (type PIMSM_JOINSPT id 156 from 9 to 8 (data null))
...
5.6 => SQ (type PIMSM_JOINSPT id 201 from 8 to 4 (data null))
...
6.2 => SQ (type PIMSM_PRUNERPBIT id 208 from 4 to 5 (data null))
...
6.8 => SQ (type PIMSM_PRUNERPBIT id 221 from 5 to 6 (data null))
7.1 => SQ (type PIMSM_JOINSPT id 226 from 8 to 4 (data null))
...
7.4 => SQ (type PIMSM_PRUNERPT id 249 from 6 to 5 (data null))
```

```
...
7.8 => SQ (type PIMSM_PRUNERPBIT id 262 from 4 to 5 (data null))
...
8.0 => SQ (type PIMSM_PRUNERPT id 285 from 5 to 4 (data null))
8.4 => SQ (type PIMSM_PRUNERPBIT id 298 from 5 to 6 (data null))
...
9.0 => SQ (type PIMSM_PRUNERPT id 313 from 6 to 5 (data null))
...
9.6 => SQ (type PIMSM_PRUNERPT id 342 from 5 to 4 (data null))
```

Uma vez montada a árvore de distribuição baseada no fonte, restariam apenas as trocas de comandos `PIMSM_JOIN` entre os roteadores para *refresh* das interfaces armazenadas na `joinlist` de cada roteador, o que é feito a cada minuto. No entanto, para fins de melhor visualização de todas as mensagens trocadas entre os roteadores na montagem das árvores multicast, as mensagens de *refresh* foram enviadas a cada 13 segundos, posto que a simulação de demonstração aqui apresentada é executada em menos de 1 minuto. Lembrando, elas são enviadas pelo método `OnTimer()` da simulação, responsável pelo envio periódico dos comandos entre os roteadores, aparecendo, então, nos tempos 13.0 e 26.0. A primeira seqüência de comandos de *refresh* pode ser vista no seguinte trecho do traço:

```
13.0 => SQ (type PIMSM_JOIN id 457 from 8 to 4 (data null))
13.0 => SQ (type PIMSM_JOIN id 460 from 8 to 6 (data null))
13.0 => SQ (type PIMSM_JOIN id 463 from 9 to 8 (data null))
13.0 => SQ (type PIMSM_JOIN id 466 from 4 to 1 (data null))
```

Como no arquivo de definição da simulação foi especificado que a simulação seria finalizada com o envio de 15 mensagens pelo *host* P1, este arquivo de traço representa a totalidade das transações executadas na simulação, sendo esta limitada para facilitar a visualização da execução do protocolo na distribuição dos pacotes multicast entre os nodos da rede.

Dada a complexidade do protocolo PIM/SM, alguns aspectos não foram contemplados no experimento realizado. Diante disto, também foram realizados experimentos complementares partindo de pequenas diferenças na simulação apresentada. Inicialmente pensemos que, para a topologia utilizada na simulação anterior, P2 deixasse de assinar o grupo G1, enviando artificialmente (através de chamada do método `handleIGMPleave()`) um comando *leave* para o R5. Logo, todas as mensagens enviadas por P1 seriam apenas enviadas para P3. Observa-se, no tempo 0.2, que, em decorrência, um comando `PIMSM_PRUNE` é enviado do R5 para o roteador R3 (RP). Desta forma, verifica-se que, quando P1 envia um comando no início da simulação, o roteador R5 envia o pacote somente para o R6, que, por sua vez, entrega-o para P3 no tempo 3.0, ficando P2 fora da árvore de distribuição. Isto é mostrado na seqüência de traço abaixo, sendo que este não representa a totalidade da simulação, dado sua similaridade com o experimento já apresentado.

```
0.0 => SQ (type PIMSM_JOIN id 2 from 9 to 8 (data null))
0.0 => SQ (type PIMSM_JOIN id 5 from 8 to 6 (data null))
0.0 => SQ (type ROUTER id 10 from 1 to 4 (data (type APP id 9 from 1 to -1 (data 0))))
0.2 => SQ (type PIMSM_PRUNE id 8 from 8 to 6 (data null))
0.6 => SQ (type ROUTER id 16 from 4 to 5 (data (type PIMSM_REGISTER id 15 from 4 to 6
(data (type APP id 9 from 1 to -1 (data 0))))))
0.6 => SQ (type PIMSM_JOIN id 19 from 8 to 6 (data null))
0.7 => SQ (type ROUTER id 21 from 6 to 8 (data (type PIMSM_REGISTERSTOP id 20 from 6 to 8
(data null))))
1.2 => SQ (type PIMSM_REGISTER id 15 from 4 to 6 (data (type APP id 9 from 1 to -1 (data 0))))
1.8 => SQ (type ROUTER id 30 from 6 to 8 (data (type APP id 29 from 1 to -1 (data 0))))
1.8 => SQ (type PIMSM_JOINRPT id 33 from 6 to 5 (data null))
...
```

```
2.4 => SQ (type ROUTER id 43 from 8 to 9 (data (type APP id 42 from 1 to -1 (data 0))))
2.4 => SQ (type PIMSM_JOINRPT id 50 from 5 to 4 (data null))
2.5 => SQ (type PIMSM_REGISTERSTOP id 34 from 6 to 4 (data null))
...
3.0 => SQ (type ROUTER id 70 from 9 to 3 (data (type APP id 69 from 1 to -1 (data 0))))
...
```

Outra análise necessária é o caso onde nenhum *host* tenha assinado dado grupo multicast junto ao R3 (RP). Desta forma, quando P1 envia algum pacote para o R3, ele sempre é consumido, não sendo enviado a nenhum outro nodo. Um novo pacote somente é transmitido quando houver uma nova transmissão feita pelo *host* P1, o que é feito nos tempos 1.2 e 4.0 da simulação. Isto pode ser visualizado na seqüência a seguir:

```
0.0 => SQ (type ROUTER id 1 from 1 to 4 (data (type APP id 0 from 1 to -1 (data 0))))
0.6 => SQ (type ROUTER id 7 from 4 to 5 (data (type PIMSM_REGISTER id 6 from 4 to 6
(data (type APP id 0 from 1 to -1 (data 0))))))
1.2 => SQ (type PIMSM_REGISTER id 6 from 4 to 6 (data (type APP id 0 from 1 to -1 (data 0))))
2.0 => SQ (type ROUTER id 11 from 1 to 4 (data (type APP id 10 from 1 to -1 (data 1))))
2.6 => SQ (type ROUTER id 17 from 4 to 5 (data (type PIMSM_REGISTER id 16 from 4 to 6
(data (type APP id 10 from 1 to -1 (data 1))))))
3.2 => SQ (type PIMSM_REGISTER id 16 from 4 to 6 (data (type APP id 10 from 1 to -1 (dat a 1))))
4.0 => SQ (type ROUTER id 21 from 1 to 4 (data (type APP id 20 from 1 to -1 (data 2))))
4.6 => SQ (type ROUTER id 27 from 4 to 5 (data (type PIMSM_REGISTER id 26 from 4 to 6
(data (type APP id 20 from 1 to -1 (data 2))))))
5.2 => SQ (type PIMSM_REGISTER id 26 from 4 to 6 (data (type APP id 20 from 1 to -1 (data 2))))
...
```

Da mesma forma, o contrário pode ocorrer, ou seja, *hosts* receptores podem assinar o RP e nenhuma mensagem ser transmitida para o grupo. Neste caso, somente os comandos `PIMSM_JOIN` de associação no RP, como mostrado no tempo 0, e as mensagens de *refresh* trafegam na rede, conforme mostram as mensagens nos tempos 13.0, 26.0 e 39.0 do trecho do traço apresentado. Observe-se que o atributo `joinDiscardInterval` permanece configurado para 13 segundos.

```
0.0 => SQ (type PIMSM_JOIN id 2 from 9 to 8 (data null))
0.0 => SQ (type PIMSM_JOIN id 5 from 8 to 6 (data null))
13.0 => SQ (type PIMSM_JOIN id 10 from 9 to 8 (data null))
13.0 => SQ (type PIMSM_JOIN id 13 from 8 to 6 (data null))
26.0 => SQ (type PIMSM_JOIN id 18 from 9 to 8 (data null))
26.0 => SQ (type PIMSM_JOIN id 21 from 8 to 6 (data null))
39.0 => SQ (type PIMSM_JOIN id 26 from 9 to 8 (data null))
39.0 => SQ (type PIMSM_JOIN id 29 from 8 to 6 (data null))
...
```

Estas simulações adicionais complementam o experimento relativo ao protocolo PIM/SM.

7.5 Comentários Adicionais

Os experimentos realizados tiveram por objetivo mostrar a dinâmica do roteamento integrante à nova arquitetura do SIMMCAST, e demonstraram a funcionalidade da ferramenta e sua fidelidade aos conceitos de cada protocolo. Os arquivos de saída das simulações permitiram a visualização da execução dos protocolos durante a troca de pacotes de controle para obtenção de informações de roteamento dos pacotes, bem como da transmissão e recepção de pacotes de dados.

Apesar de não ser parte integrante da implementação de roteamento no simulador, a aplicação utilizada nos experimentos foi apresentada sob forma de diagrama UML,

buscando viabilizar a correlação entre classes e métodos referenciados pelos arquivos de definição das simulações. Comentários referentes à sintaxe dos arquivos de configuração e de saída da simulação foram apresentados, no intuito de oferecer uma referência para novas simulações desenvolvidas com o SIMMCAST.

Capítulo 8

CONCLUSÃO

A ferramenta SIMMCAST é ao mesmo tempo o resultado prático e a prova-de-conceito de um projeto de pesquisa cujo objetivo é investigar a simulação de protocolos multicast e sistemas distribuídos baseados em grupo. Previamente, foi apresentada uma revisão bibliográfica referente à roteamento multicast e simuladores de redes. Na seqüência foram descritos aspectos fundamentais do SIMMCAST, como seu *framework* e blocos básicos de construção, fornecendo, ainda, a descrição da implementação do simulador sob forma de diagramas de classes no padrão UML.

Este trabalho abordou o projeto de facilidades de roteamento multicast, necessárias em simulações mais detalhadas, baseado na combinação dos blocos. A extensibilidade do *framework* nos permitiu agregar roteamento ao SIMMCAST via nodos roteadores. Os benefícios em termos de ferramenta são dois: (a) permitir trabalhar com protocolos multicast e sistemas distribuídos baseados em comunicação em grupo usando topologias físicas arbitrárias, com distribuição de pacotes via árvore multicast; e (b) auxiliar o desenvolvimento, através do estudo, análise e comparação, de protocolos de roteamento multicast.

Os modelos de roteamento desenvolvidos para o SIMMCAST e apresentados nas seções anteriores, concebidos de acordo com a filosofia incremental que rege o simulador, permitem a evolução gradual do experimento do abstrato para o detalhado, através da exploração das diferentes abordagens providas pelo modelo de blocos *host* e *router*: sem roteamento, com roteamento estático e com roteamento dinâmico, exceto quando o mesmo envolve a investigação de protocolos de roteamento.

O desenvolvimento dos protocolos de roteamento PIM, para ambientes denso e esparso, viabilizam desde já a realização de experimentos que necessitam roteamento dinâmico subjacente, onde a troca de mensagens de controle entre os roteadores é fator importante na avaliação do experimento. Para a investigação de protocolos de roteamento, os protocolos desenvolvidos abrem espaço para o desenvolvimento de outros, sejam intra ou inter-domínios, e portanto, tiveram suas implementações também apresentadas sob forma de diagramas de classes UML, explicitando os elementos onde foram aplicadas técnicas de herança e composição do *framework*. Além disso, o detalhamento dos arquivos de definição e saída das simulações visa mostrar como devem ser gerados e avaliados, respectivamente, estes arquivos de suma importância na execução de simulações com o SIMMCAST.

O simulador SIMMCAST é um projeto de pesquisa em andamento. Novas características estão sendo desenvolvidas para o simulador, entre as quais: suporte ao uso de portas, através da criação de diferentes pontos de chegadas de pacotes em um nodo; e importação de topologias descritas em arquivos e geradas por geradores automáticos de topologias.

Trabalhos futuros com o simulador estão relacionados a modelagem de falhas, temporárias ou permanentes, de *hosts*, roteadores ou *links*, bem como ao desenvolvimento de protocolos de roteamento inter-domínio. Além disso, a ferramenta pode ser aumentada com uma série de facilidades, como o desenvolvimento de interface gráfica.

Acreditamos que os princípios utilizados na elaboração do SIMMCAST, através de seu *framework* de simulação e blocos de construção, podem ser utilizados com sucesso também em projetos com outros enfoques, tal como em aplicações durante um processo de avaliação de desempenho, realizado por outros pesquisadores.

Resultados parciais deste trabalho foram previamente apresentados sob forma de artigo, com o título "Simulação de Roteamento na Avaliação de Protocolos Multicast e Sistemas Distribuídos de Grupo", no I Workshop de Performance, evento integrante do XXII Simpósio Brasileiro de Computação, em julho de 2002 ([45]).

Apêndice A

CÓDIGO FONTE DAS IMPLEMENTAÇÕES PIM

A.1 Arquivos Comuns às Implementações PIM/DM e PIM/SM

A.1.1 PIMInterfaceList.java

```
package simmcast.route;

import java.util.*;
import java.io.*;


/**
 * This class implements a list of interfaces.
 * In PIM/DM, it stores all nodes that send prune messages to router.
 * In PIM/SM, it stores all nodes that send join messages to router RP.
 * If sourcer = 0 than means any source (*)
 */

public class PIMInterfaceList extends Hashtable {
   public Vector getInterfaces(int source_, int group_) {
      return (Vector)(get(new PIMInterfaceListKey(source_, group_)) );
   }

   public int getInterfacesSize(int source_, int group_) {
      Vector interfaces = (Vector) get(new PIMInterfaceListKey(source_, group_));
      return interfaces == null ? 0 : interfaces.size();
   }

   public void addInterface(int source_, int group_, int interface_) {
      PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
      Vector list = (Vector)( get(key) );
      if (list == null) {
         list = new Vector();
         put(key, list);
      }
      Integer i = new Integer(interface_);
      if (list.indexOf(i) == -1)
         list.addElement(i);
      }
   }

   public void removeAllInterfaces(int source_, int group_) {
      PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
      remove(key);
   }


   public void removeInterface(int source_, int group_, int interface_) {
      PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
      Vector list = (Vector)( get(key) );
      if (list == null)
```

```
            return;
         list.removeElement(new Integer(interface_));
      }


      public boolean isInterfacePresent(int source_, int group_, int interface_)
      {
         PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
         Vector list = (Vector)( get(key) );
         if (list == null)
            return false;
         if (interface_ == 0)            // it's not important the interface
            return true;
         return (list.indexOf(new Integer(interface_)) != -1);
      }
   }
```

A.1.2 PIMInterfaceListKey.java

```
package simmcast.route;

public class PIMInterfaceListKey {

   public int source;
   public int group;

   public PIMInterfaceListKey(int source_, int group_) {
      source = source_;
      group = group_;
   }

   public int hashCode() {
      return source + group * 1000000;
   }

   public boolean equals(Object o_) {
      if (o_ instanceof PIMInterfaceListKey) {
         PIMInterfaceListKey ilk = (PIMInterfaceListKey)o_;
         return ( (source == ilk.source) && (group == ilk.group) );
      } else
         return false;
   }
}
```

A.1.3 PIMPacket.java

```
package simmcast.route;

import simmcast.network.*;

public class PIMPacket extends Packet implements MulticastControlPacket {

   int fromHost;
   int toHost;

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */

   public PIMPacket(int from_, int to_, PacketType type_, int size_,
                    int fromHost_, int toHost_) {
      super(from_, to_, type_, size_);
      fromHost = fromHost_;
      toHost = toHost_;
   }
}
```

A.2 Arquivos Exclusivos da Implementação do PIM/DM

A.2.1 PIMDMNode.java

```
package simmcast.route;

import java.util.*;
import java.io.*;

/**
 * This class implement the PIMDM Node. The PIMDM Node defines
 * a prune list table that related each source node to multicast
 * group destination.
 * The prune list is discart according to interval defined by user
 * on the script file from simulation.
 */


public class PIMDMNode extends DefaultRouterNode {

   /**
    * This is a table indicating networkIds of nodes that answered with
    * a prune message, according to a given destination.
    */
   PIMInterfaceList pruneList = new PIMInterfaceList();

   float pruneDiscardInterval, timeoutGraft;

   boolean isReportSolicited = false;
   boolean isLeaveSolicited  = false;
   int     hostIGMP;
   int     groupIGMP;

   public void setPruneDiscardInterval(float time_) {
      System.out.println("Set pruneDiscardInterval =  ["+time_+"] !");
      pruneDiscardInterval = time_;
   }

   public void setTimeoutGraft(float time_) {
      System.out.println("Set timeoutGraft =  ["+time_+"] !");
      pruneDiscardInterval = time_;
   }

   public void handleIGMPReport(int group_, int host_) {
      isReportSolicited = true;
      groupIGMP = group_;
      hostIGMP = host_;
   }

   public void handleIGMPLeave(int group_, int host_) {
      isLeaveSolicited = true;
      groupIGMP = group_;
      hostIGMP = host_;
   }
}
```

A.2.2 PIMDMAlgorithmStrategy.java

```
package simmcast.route;

import java.util.*;
import java.io.*;

import simmcast.node.*;
import simmcast.network.*;
import simmcast.trace.*;


/**
 * This class implement the PIMDMReceiver.
```

```
 * When a message packet is receive, the type message is checked.
 * According to the type message the logical protocol is actived.
 */

// extra class to help on graft command
class GraftInformation {
    int destinationInterface;
    int sourceAddress;
    int destinationAddress;

    public boolean equals(Object o_) {
       GraftInformation gi = (GraftInformation)o_;
       return ( (gi.destinationInterface == destinationInterface)
             && (gi.sourceAddress == sourceAddress)
             && (gi.destinationAddress == destinationAddress) );
    }
}


public class PIMDMAlgorithmStrategy extends RoutingAlgorithmStrategy {

    public static final PacketType RPF_TYPE = new PacketType("RPF");

    // size of commands packages: use size = 104 to use the same band wide
    // of PIM experiments

    public static final int TAMBYTESCMD = 104;
    public static final int ANYSOURCE = 0;

    public static final TimerIdentifier pruneTimer = new TimerIdentifier();
    public static final TimerIdentifier graftAckTimer = new TimerIdentifier();

    PIMDMNode router;

    Vector pendingAcks = new Vector();

    StaticAlgorithmStrategy staticAlgorithm;

    public PIMDMAlgorithmStrategy(DefaultRouterNode router_,
                          StaticAlgorithmStrategy staticAlgorithm_) {
       super(router_);
       staticAlgorithm = staticAlgorithm_;
       router = (PIMDMNode)router_;
    }

    /**
      * Start Timer
      */

    public void begin() {
       setTimer(router.pruneDiscardInterval, pruneTimer);
       setTimer(timeoutGraft, graftAckTimer);
       System.out.println("PIMDMReceiver ["+router.getNetworkId()+"] Started!");
    }

    /**
      * Multicast Control Group
      */

    public void onExecute() {
       int thisNode = router.getNetworkId();
       // take the neighbors list
       int [] neighbors = router.neighborRouterIds();

       // verify if there is an igmp report command pending
       if (router.isReportSolicited) {
          router.isReportSolicited = false;
          // remove from list
          router.pruneList.removeInterface(ANYSOURCE,router.groupIGMP, router.hostIGMP);
          // and verify the neighbors
          for (int i=0; i< router.neighborRouterCount();i++) {
             sendToNeighbor(new PIMDMGraftPacket(thisNode, neighbors[i],
                           TAMBYTESCMD, ANYSOURCE, router.groupIGMP));
```

```
         // start timer until receive an GraftAck
         GraftInformation graft = new GraftInformation();
         graft.destinationInterface = neighbors[i];
         graft.sourceAddress = ANYSOURCE;
         graft.destinationAddress = router.groupIGMP;
         pendingAcks.add(graft);
      }
      return;
   }

   // verify if there is an igmp leave command pending
   if (router.isLeaveSolicited) {
      router.isLeaveSolicited = false;
      // verify if already have an entry on prunelist
      if (!(router.pruneList.isInterfacePresent(ANYSOURCE, router.groupIGMP,
            router.hostIGMP))) {
         // add entry on the prunelist
         router.pruneList.addInterface(ANYSOURCE, router.groupIGMP,router.hostIGMP );
         // and verify the neighbors
         for (int i=0; i< router.neighborRouterCount();i++) {
            sendToNeighbor(new PIMDMPrunePacket(thisNode, neighbors[i],
                           TAMBYTESCMD, ANYSOURCE, router.groupIGMP));
         }
      }
   }
}

/**
  * Message Control Interface
  */

public void handleControlPacket(Packet p_) {
   // set local variables from packet
   int sourceInterface = p_.getSource();
   int thisNode = router.getNetworkId();

   // take information on the data area
   int sourceAddress = ((PIMPacket)p_).fromHost;
   int destinationAddress = ((PIMPacket)p_).toHost;

   boolean isInterfacetoGroup = false;

   // take the interface that received the packet
   int pathOptimum = 0;
   if (sourceAddress != ANYSOURCE)
      pathOptimum = (staticAlgorithm.getNextHops (new Packet(thisNode, sourceAddress,
                    RPF_TYPE, 0, new Packet(thisNode, sourceAddress, RPF_TYPE, 0))))[0];
   // take the neighbors list
   int [] neighbors = router.neighborRouterIds();

   // check if the received message is a PRUNE
   if (p_ instanceof PIMDMPrunePacket) {
      // verify if this router is interface to a group
      int [] neighborHosts = router.getNeighborHostsInGroup(destinationAddress);
      int[] nextHosts = new int[neighborHosts.length];
      int nextItem = 0;
      if (nextHosts.length > 0)
         isInterfacetoGroup = true;
      else
         isInterfacetoGroup = false;

      // check if it is the optimum path
      if (isInterfacetoGroup && sourceInterface == pathOptimum) {
         // send an Join command to continue on the tree
         sendToNeighbor(new PIMDMJoinPacket(thisNode, sourceInterface,
                        TAMBYTESCMD, sourceAddress, destinationAddress));
      } else {
         // then add interface to the pruneList
         router.pruneList.addInterface(sourceAddress, destinationAddress, sourceInterface);
      }
      return;
   }
```

```
   // check if the received message is a JOIN
   if (p_ instanceof PIMDMJoinPacket) {
      // if is than remove entry from the pruneList
      if (sourceInterface == pathOptimum)
         router.pruneList.removeInterface(sourceAddress, destinationAddress,
                                          sourceInterface);
      return;
   }

   // check if the received message is a GRAFT
   if (p_ instanceof PIMDMGraftPacket) {
      // send an ACK through the interface to source
      sendToNeighbor(new PIMDMGraftAckPacket(thisNode,sourceInterface,
                     TAMBYTESCMD, sourceAddress, destinationAddress));
      // check the pruneList
      if (sourceInterface == pathOptimum) {
         if (router.prunelist.isInterfacePresent (sourceAddress, destinationAddress,
            sourceInterface)) {
            router.pruneList.removeInterface(sourceAddress, destinationAddress,
                                             sourceInterface);
           // forward the command through the tree
           for (int i=0; i< router.neighborRouterCount();i++) {
              if (neighbors[i] == sourceInterface)
                 continue;
           if (router.pruneList.isInterfacePresent(sourceAddress, destinationAddress,
               neighbors[i])) {
              // then send the command to interface
              sendToNeighbor(new PIMDMGraftPacket(thisNode,neighbors[i],
                             TAMBYTESCMD, sourceAddress, destinationAddress));
              // start timer until receive an GraftAck
              GraftInformation graft = new GraftInformation();
              graft.destinationInterface = neighbors[i];
              graft.sourceAddress = sourceAddress;
              graft.destinationAddress = destinationAddress;
              pendingAcks.add(graft);
           }
         }
      }
      return;
   }

   // check if the received message is a GRAFTACK
   if (p_ instanceof PIMDMGraftAckPacket) {
      // finish timer when receive an GraftAck
      GraftInformation graft = new GraftInformation();
      graft.destinationInterface = sourceInterface;
      graft.sourceAddress = sourceAddress;
      graft.destinationAddress = destinationAddress;
      pendingAcks.remove(graft);
      return;
   }
}

/**
  * Routine that implements PIM/DM forwarding packets
  */

public int[] getNextHops(Packet p_) {

   // set local variables from packet
   int sourceInterface = p_.getSource();
   int thisNode = router.getNetworkId();

   // take information on the data area
   Packet packetReceived = (Packet) (p_.getData());
   int sourceAddress = packetReceived.getSource();
   int destinationAddress = packetReceived.getDestination();

   // to forwarding the message verify on the unicast routing table
   // which is the optimum path
   // if the packet was received by interface used to send unicast
   // messages to source node. If not discard the packet. Otherelse
   // send the copies of the packet to all interfaces that no one
```

```
// message prune was received

// take the interface that received the packet
int pathOptimum = (staticAlgorithm.getNextHops (new Packet(thisNode,
   sourceAddress, RPF_TYPE, 0, new Packet(thisNode, sourceAddress, RPF_TYPE, 0))))[0];

// verify if router receive report/leave from host
onExecute();

// check if it is the optimum path
if (pathOptimum != sourceInterface)
{
   // discard the message if was not received by optimum path
   // and send a prune message by interface that received the package
   // entry: int from_, int to_, PacketType type_, int size_
    sendToNeighbor(new PIMDMPrunePacket(thisNode, sourceInterface,
                   TAMBYTESCMD, sourceAddress, destinationAddress));
   return new int[0];
}
// otherwise send copies of the message to all interfaces that are
// not on the pruneList

// take the neighbors list
int [] neighbors = router.neighborRouterIds();

boolean thereIsInterface = false;

int[] nextHops = new int[neighbors.length];
int nextItem = 0;
// check to see which was the received interface
for (int i=0; i< router.neighborRouterCount();i++) {
   // check if it is not the source interface
   // if yes abort send
   if (neighbors[i] == sourceInterface)
      continue;

   // check if the node is on the pruneList
   // if yes abort send
   if (router.pruneList.isInterfacePresent(sourceAddress, destinationAddress,
       neighbors[i]) && router.pruneList.isInterfacePresent(ANYSOURCE,
       destinationAddress, neighbors[i]))
      continue;

     // set that at least to one interface the package will be sent
     thereIsInterface = true;

     // otherwise send the packet thought this interface to destinationAddress
     nextHops[nextItem] = neighbors[i];
     nextItem++;
}


int [] neighborHosts = router.getNeighborHostsInGroup(destinationAddress);
// check to see which didn't do leave
for (int i=0; i< neighborHosts.length;i++) {
   // check if it is not the source interface
   // if yes abort send
   if (neighborHosts[i] == sourceInterface)
      continue;

   // check if the node is on the pruneList
   // if yes abort send
   if (router.pruneList.isInterfacePresent(ANYSOURCE, destinationAddress, neighbors[i]))
      continue;

   // set that at least to one interface the package will be sent
   thereIsInterface = true;

   // otherwise send the packet thought this interface to destinationAddress
   nextHops[nextItem] = neighborHosts[i];
   nextItem++;
}
```

```
            // if there is no more interfaces to send package, then send prune
            if (!thereIsInterface) {
               // entry: int from_, int to_, PacketType type_, int size_
               sendToNeighbor(new PIMDMPrunePacket(thisNode, sourceInterface,
                              TAMBYTESCMD, sourceAddress, destinationAddress));
            }

            int [] result = new int[nextItem];
            if (nextItem > 0)
               System.arraycopy(nextHops, 0, result, 0, nextItem);
            return result;
         }

         public void onTimer(Object o_) {

            if (o_ == pruneTimer) {
               router.pruneList = new PIMInterfaceList();
               setTimer(router.pruneDiscardInterval, null);
            } else {
               // check if ack until not received to an Graft command
               if (pendingAcks.size() > 0) {
                  for (int i = 0; i < pendingAcks.size(); i++) {
                     GraftInformation graft = (GraftInformation)(pendingAcks.get(i));
                     sendToNeighbor(new PIMDMGraftPacket(router.getNetworkId(),graft.destinationInterface,
                                    TAMBYTESCMD, graft.sourceAddress, graft.destinationAddress));
                  }
                  pendingAcks.clear();
               }
               setTimer(router.timeoutGraft, graftAckTimer);
            }
         }
      }
```

A.2.3 PIMDMJoinPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a Join command from PIMDM
 */

public class PIMDMJoinPacket extends PIMPacket {

   public static final PacketType PACKET_TYPE = new PacketType("PIMDM_JOIN");

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */

   public PIMDMJoinPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
      super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
   }
}
```

A.2.4 PIMDMPrunePacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a Prune command from PIMDM
 */

public class PIMDMPrunePacket extends PIMPacket {
```

```
    public static final PacketType PACKET_TYPE = new PacketType("PIMDM_PRUNE");

    /**
     * Constructs a new packet with the specified characteristics.
     * No consistencies are made on the validity of the entered data.
     */

    public PIMDMPrunePacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
       super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
    }
}
```

A.2.5 PIMDMGraftPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a Graft command from PIMDM
 */

public class PIMDMGraftPacket extends PIMPacket {

    public static final PacketType PACKET_TYPE = new PacketType("PIMDM_GRAFT");

    /**
     * Constructs a new packet with the specified characteristics.
     * No consistencies are made on the validity of the entered data.
     */

    public PIMDMGraftPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
       super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
    }
}
```

A.2.6 PIMDMGraftAckPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a Graft command from PIMDM
 */

public class PIMDMGraftAckPacket extends PIMPacket {

    public static final PacketType PACKET_TYPE = new PacketType("PIMDM_GRAFTACK");

    /**
     * Constructs a new packet with the specified characteristics.
     * No consistencies are made on the validity of the entered data.
     */

    public PIMDMGraftAckPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
       super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
    }
}
```

A.3 Arquivos Exclusivos da Implementação do PIM/SM

A.3.1 PIMSMNode.java

```
package simmcast.route;

import java.util.*;
import java.io.*;

/**
 * This class implement the PIMSM Node.
 * The PIMSM Node defines an RendezVous Point RP to each multicast group
 * that receive messages from the sender and forward
 * the messages to destinations nodes, according to the
 * selected group.
 * The list RP is selected by user on the simulation file .SIM.
 * There is one RP to each multicast group.
 * The PIMSM node defines an list from nodes that sent join to RP.
 * The join list is discart according to interval defined by user
 * on the configure file from simulation.
 */

public class PIMSMNode extends DefaultRouterNode {

   /**
    * This is a table indicating networkIds of nodes that sent
    * a join message, according to a given destination.
    */

   PIMInterfaceList  joinList = new PIMInterfaceList();
   PIMSMRendezVousPointList groupRPList = new PIMSMRendezVousPointList();
   PIMSMMessagesCounterList messageCounterList = new PIMSMMessagesCounterList();
   PIMSMList kindOfTreeList = new PIMSMList();
   PIMSMList kindOfSourceList = new PIMSMList();
   PIMSMTimeGroupList  timerCounterList = new PIMSMTimeGroupList();

   float joinDiscardInterval = 180;      // time default: 3 min (60sec*3)
   int   threshold;

   boolean isReportSolicited = false;
   boolean isLeaveSolicited  = false;
   int     hostIGMP;
   int     groupIGMP;

   public void setJoinDiscardInterval(float time_) {
      System.out.println("Set JoinPruneDiscardInterval = ["+time_+"] !");
      joinDiscardInterval = time_;
   }

   public void setThreshold(int threshold_) {
      System.out.println("Set Threshold =  ["+threshold_+"] !");
      threshold = threshold_;
   }

   public void setRendezVousPoint(int group_, int nodeRP_) {
      System.out.println("Set Group = ["+group_+"] --> RendezVous Point = ["+nodeRP_+"] !");
      groupRPList.addRendezVousPoint(group_, nodeRP_);
   }

   public void handleIGMPReport(int group_, int host_) {
      isReportSolicited = true;
      groupIGMP = group_;
      hostIGMP = host_;
   }

   public void handleIGMPLeave(int group_, int host_) {
      isLeaveSolicited = true;
      groupIGMP = group_;
      hostIGMP = host_;
   }
}
```

A.3.2 PIMSMAlgorithmStrategy.java

```
package simmcast.route;

import java.util.*;
import java.io.*;

import simmcast.node.*;
import simmcast.network.*;
import simmcast.trace.*;

/**
 * This class implement the PIMSM Receiver.
 * When a message packet is receive, the type message is checked.
 * According to the type message the logical protocol is actived.
 */

class TimerIdentifier {
}

public class PIMSMAlgorithmStrategy extends RoutingAlgorithmStrategy {

    public static final PacketType UNICAST_TYPE = new PacketType("RPF");

    public static final int TAMBYTESCMD = 104;
    public static final int ANYSOURCE = 0;

    public static final int RPTTREE = 1;
    public static final int SPTTREE = 2;

    public static final int REGISTER_ACTIVE = 1;
    public static final int TREE_ACTIVE = 2;

    public static final int TIME_INFINITO = -1;

    public static final TimerIdentifier joinTimer = new TimerIdentifier();
    public static final TimerIdentifier refreshTimer = new TimerIdentifier();

    PIMSMNode router;
    StaticAlgorithmStrategy staticAlgorithm;


    public PIMSMAlgorithmStrategy(DefaultRouterNode router_,
                                  StaticAlgorithmStrategy staticAlgorithm_) {
       super(router_);
       staticAlgorithm = staticAlgorithm_;
       router = (PIMSMNode)router_;
    }

    /**
      * Start Timers
      */

    public void begin() {

       setTimer(router.joinDiscardInterval, joinTimer);
       setTimer(60, refreshTimer);            // on each min router refresh join
       System.out.println("PIMSMReceiver ["+router.getNetworkId()+"] Started!");
    }

    /**
      * Multicast Control Group
      */

    public void onExecute() {
       // take router id
       int thisNode = router.getNetworkId();
       // take the RP from group
       int routerRPfromGroup, pathRPOptimum;

       // verify if there is an igmp report command pending
       if (router.isReportSolicited) {
          router.isReportSolicited = false;
```

```
      routerRPfromGroup = router.groupRPList.getRendezVousPoint(router.groupIGMP);
      // take the unicast path to RP
      pathRPOptimum = (staticAlgorithm.getNextHops (new Packet(thisNode,
                      routerRPfromGroup, UNICAST_TYPE, 0,
                      new Packet(thisNode, routerRPfromGroup, UNICAST_TYPE, 0))))[0];

       // verify if already have an (*,G) entry on joinlist
      if (!(router.joinList.isInterfacePresent(ANYSOURCE, router.groupIGMP,
            router.hostIGMP))) {
         // if is not present: add router source on list
         router.kindOfTreeList.addSGOnList(ANYSOURCE,router.groupIGMP, RPTTREE);
         //if report => entry isn't remove
         router.timerCounterList.addSGroupOnList(ANYSOURCE, router.groupIGMP, TIME_INFINITO);
         // and add the entry (*,G,interface) on the joinlist
         router.joinList.addInterface(ANYSOURCE,router.groupIGMP, router.hostIGMP);
      }
      // any case send a Join message to RP
      sendToNeighbor(new PIMSMJoinPacket(thisNode, pathRPOptimum,
                     TAMBYTESCMD, ANYSOURCE, router.groupIGMP));
   }

   if (router.isLeaveSolicited) {
      router.isLeaveSolicited = false;
       routerRPfromGroup = router.groupRPList.getRendezVousPoint(router.groupIGMP);
      // take the unicast path to RP
      pathRPOptimum = (staticAlgorithm.getNextHops (new Packet(thisNode,
                      routerRPfromGroup, UNICAST_TYPE, 0,
                      new Packet(thisNode, routerRPfromGroup, UNICAST_TYPE, 0))))[0];

      if (router.joinList.isInterfacePresent(ANYSOURCE, router.groupIGMP,
          router.hostIGMP)) {
         // remove the entry (*,G,interface) from the joinlist
         router.joinList.removeInterface(ANYSOURCE,router.groupIGMP, router.hostIGMP);
         // verify if became empty
         if (router.joinList.getInterfacesSize(ANYSOURCE,router.groupIGMP) == 0)
         {
            // free status from router
            // remove counter from counterlist -> SGroup
            router.kindOfTreeList.removeSGFromList(ANYSOURCE,router.groupIGMP);
            router.timerCounterList.removeSGroupFromList(ANYSOURCE, router.groupIGMP);
            // if empty send a Prune message to RP
            sendToNeighbor(new PIMSMPrunePacket(thisNode, pathRPOptimum,
                           TAMBYTESCMD, ANYSOURCE, router.groupIGMP));
         } else {
            // change time to discard time
            router.timerCounterList.setTimeFromSGroupOnList(ANYSOURCE,
                   router.groupIGMP, (int) router.joinDiscardInterval);
         }
      }
   }
}

/**
  * Message Control Interface
  */

public void handleControlPacket(Packet p_) {

   /*
    * Set enviroment variables from router
    */

   Packet packetReceived;
   int sourceInterface, thisNode, sourceAddress, destinationAddress;
   int routeRPfromGroup, kindOftree, kindOfsource;
   boolean isInterfacetoGroup, isRPfromGroup;

   sourceInterface = p_.getSource();
   thisNode = router.getNetworkId();

   if (p_ instanceof PIMSMRegisterPacket) {
      sourceAddress = ((PIMSMRegisterPacket)p_).fromHost;
```

```
   destinationAddress = ((PIMSMRegisterPacket)p_).toHost;
} else {
   sourceAddress = ((PIMPacket)p_).fromHost;
   destinationAddress = ((PIMPacket)p_).toHost;
}
// verify if this router is interface to a group from host - DR from group
isInterfacetoGroup = false;      // initialize

// verify if the router is RP from group
routeRPfromGroup = router.groupRPList.getRendezVousPoint(destinationAddress);
isRPfromGroup = false;
if (thisNode == routeRPfromGroup)
   isRPfromGroup = true;
// verify if is first-hop
if ((sourceAddress == sourceInterface) && !isRPfromGroup)
   isInterfacetoGroup = true;
else {
   int[] neighborHosts = router.getNeighborHostsInGroup(destinationAddress);
   int[] nextHosts = new int[neighborHosts.length];
   for (int i=0; i< nextHosts.length;i++) {
   if (neighborHosts[i] == sourceAddress)
      isInterfacetoGroup = true;     // router is interface to destinationAddress
   else
      continue;
   }
}

// take the treetype from receiver
kindOftree = router.kindOfTreeList.getValueFromSG(sourceAddress, destinationAddress);
if (kindOftree == 0)
   kindOftree = RPTTREE;              // if it's not on list use default value

// take the type of transmission to the group
kindOfsource = router.kindOfSourceList.getValueFromSG(sourceAddress, destinationAddress);
if (kindOfsource == 0)
   kindOfsource = REGISTER_ACTIVE;    // if it's not on list use default value

// take the unicast path to RP
int pathRPOptimum = 0;
// if router is not a RP to group
if (!isRPfromGroup)
   pathRPOptimum = (staticAlgorithm.getNextHops (new Packet(thisNode, routeRPfromGroup,
                   UNICAST_TYPE, 0, new Packet(thisNode,
                   routeRPfromGroup, UNICAST_TYPE, 0))))[0];

// take the unicast path to source S
int pathSourceOptimum = 0;
if (sourceAddress != 0)
   pathSourceOptimum = (staticAlgorithm.getNextHops (new Packet(thisNode,
                       sourceAddress, UNICAST_TYPE, 0,
                       new Packet(thisNode, sourceAddress, UNICAST_TYPE, 0))))[0];

/*
 * Check the received command
 */

// verify if is a prune command
 if (p_ instanceof PIMSMPrunePacket) {
   // remove the entry (sourceAddress, G, interface) from joinlist
   // if router is not DR
   if (!isInterfacetoGroup) {
      router.joinList.removeInterface(sourceAddress,destinationAddress, sourceInterface);

      // verify if became empty
      if (router.joinList.getInterfacesSize(sourceAddress,destinationAddress) == 0)
      {
         // free status from router
         // remove counter from counterlist -> SGroup
         router.messageCounterList.removeSGroupFromList(sourceAddress,
                                                        destinationAddress);
         // remove control from treeType -> SGroup
         router.kindOfTreeList.removeSGFromList(sourceAddress,destinationAddress);
         // remove control from sourceType -> SGroup
```

```
         router.kindOfSourceList.removeSGFromList(sourceAddress,destinationAddress);

         // verify if this node is a RP from group
         if (thisNode == routeRPfromGroup) {
            // send a registerstop to disconect SOURCE DR from treeType - Unicast message
            // unicast transmission
            send (new PIMSMRegisterStopPacket(thisNode,
                  sourceInterface,TAMBYTESCMD, sourceAddress, destinationAddress),
                  RouterPacket.MAX_TTL);
         } else {
            // then forward a prune command from router towards RP by unicast path
            sendToNeighbor(new PIMSMPrunePacket(thisNode, pathRPOptimum,
                           TAMBYTESCMD, sourceAddress, destinationAddress));
         }
      } // else // list remains with interfaces
   }
   return;
}

// verify if is a join command

if (p_ instanceof PIMSMJoinPacket) {
   // first check if interface is already on the joinlist
   if (!(router.joinList.isInterfacePresent(sourceAddress,
         destinationAddress, sourceInterface))) {
      // if false then add the source to joinlist
      router.joinList.addInterface(sourceAddress,destinationAddress, sourceInterface);
      if (sourceAddress != ANYSOURCE)
         router.joinList.addInterface(ANYSOURCE, destinationAddress, sourceInterface);

      // if router is not a RP then re-send towards RP
      if (!isRPfromGroup ) {
         // verify if this was the first entry for the pair
         if (router.joinList.getInterfacesSize(sourceAddress, destinationAddress) == 1) {
             sendToNeighbor(new PIMSMJoinPacket(thisNode, pathRPOptimum,
                           TAMBYTESCMD, sourceAddress, destinationAddress));
         } // else // if group is already present don't forward join to RP

      } // else // if is a RP router from group only add on S,G on joinlist

   } // else // interface is already present

   // set new time from entry to refresh control
   // if entry is not a report than refresh entry
   if (router.timerCounterList.getTimeFromSGroup (sourceAddress,destinationAddress)
      != TIME_INFINITO)
      // restart timer - join received
      router.timerCounterList.setTimeFromSGroupOnList(sourceAddress, destinationAddress,
                                                      (int) router.joinDiscardInterval);
      return;
}

// verify if is a pruneSPT command: command send by RP to S

if (p_ instanceof PIMSMPruneSPTPacket) {
   // first check if interface is already on the joinlist
   if (router.joinList.isInterfacePresent(sourceAddress,
       destinationAddress, sourceInterface)) {
      // if true then remove the interface to source from joinlist
      router.joinList.removeInterface(sourceAddress,
             destinationAddress, sourceInterface);

      // if router is not a DR from source S, forward command by unicast path to S
      if (!isInterfacetoGroup) {
         // verify if there is no more interfaces to group destinationAddress
          if (router.joinList.getInterfacesSize(sourceAddress, destinationAddress) == 0) {
            sendToNeighbor(new PIMSMPruneSPTPacket(thisNode, pathSourceOptimum,
                           TAMBYTESCMD, sourceAddress, destinationAddress));
         }
      } // else // if is DR from group can't process prunespt command
   }
   return;
}
```

```
// verify if is a joinSPT command

if (p_ instanceof PIMSMJoinSPTPacket) {
      // first check if interface is already on the joinlist
      router.joinList.addInterface(sourceAddress, destinationAddress, sourceInterface);
      // if router is not a DR from source S send by unicast path to S
      if (!isInterfacetoGroup) {
         // verify if this was the first entry for the pair
         if (router.joinList.getInterfacesSize(sourceAddress, destinationAddress) == 1) {
            // take the unicast path to source
            // forward the command
            sendToNeighbor(new PIMSMJoinSPTPacket(thisNode, pathSourceOptimum,
                           TAMBYTESCMD, sourceAddress, destinationAddress));
         }
      } else {
         // if router is DR then change the treeType to RPT
         router.kindOfTreeList.setValueToSGOnList(sourceAddress, destinationAddress, SPTTREE);
         // verify if there is two tree - SPT and RPT and then prune RPT
         if (pathSourceOptimum != pathRPOptimum) {
            sendToNeighbor(new PIMSMPruneRPBitPacket(thisNode, pathRPOptimum,
                           TAMBYTESCMD, sourceAddress, destinationAddress));
         }
      }
      return;
}

// verify if is a PIMRegister Unicast command
if (p_ instanceof PIMSMRegisterPacket) {
   if (isRPfromGroup) {
      // verify if there is a shared tree (router.ANYSOURCE)for group destinationAddress
      if (router.joinList.getInterfacesSize(ANYSOURCE, destinationAddress) > 0) {
         // first take data packet and send to receivers
         Packet packetbyRegister = (Packet) (p_.getData());
         // only sent if data is valid
         if (packetbyRegister == null) {
            return;
         }
         // distributed by RPT
         send (packetbyRegister, RouterPacket.MAX_TTL);

         // then RP send an JoinRPT hop-by-hop until Source S: use unicast path from RP to S
         sendToNeighbor(new PIMSMJoinRPTPacket(thisNode, pathSourceOptimum,
                        TAMBYTESCMD, sourceAddress, destinationAddress));
         send (new PIMSMRegisterStopPacket(thisNode, sourceInterface, TAMBYTESCMD,
               sourceAddress, destinationAddress),RouterPacket.MAX_TTL);
      }
   }
   return;
}

// verify if is a PIMRegisterStop Unicast command

if (p_ instanceof PIMSMRegisterStopPacket) {
   // set to use shared tree RPT and stop to use register unicast to send messages to RP
   router.kindOfSourceList.setValueToSGOnList(sourceAddress, destinationAddress, TREE_ACTIVE);
   return;
}

// verify if is a joinRPT command

if (p_ instanceof PIMSMJoinRPTPacket) {
   if (isInterfacetoGroup) {
      // if router is a DR from group S then change to RPT tree
      // first verify if node is already SPTTREE
      if (router.kindOfTreeList.getValueFromSG(sourceAddress,destinationAddress) != SPTTREE)
         router.kindOfTreeList.setValueToSGOnList(sourceAddress, destinationAddress, RPTTREE);
   }
   else {
      // forward command by unicast path
      // add interface to joinlist (S,G,i)
      router.joinList.addInterface(sourceAddress, destinationAddress, sourceInterface);
      sendToNeighbor(new PIMSMJoinRPTPacket(thisNode, pathSourceOptimum,
```

```
                        TAMBYTESCMD,sourceAddress, destinationAddress));
      }
      return;
   }

   // verify if is a pruneRPT command

   if (p_ instanceof PIMSMPruneRPTPacket) {
      if (!isInterfacetoGroup) {
         // if router is not a DR from group S remove (*,G) from join: all interfaces
         router.joinList.removeAllInterfaces(sourceAddress, destinationAddress);
         sendToNeighbor(new PIMSMPruneRPTPacket(thisNode, pathSourceOptimum, TAMBYTESCMD,
            sourceAddress, destinationAddress));
      } else {
         // if a DR receives a pruneRPT command disconect from shared tree
         router.joinList.removeInterface(sourceAddress, destinationAddress,sourceInterface);
      }
      return;
   }

   // verify if is a pruneRPBit command

   if (p_ instanceof PIMSMPruneRPBitPacket) {
      if (pathRPOptimum != pathSourceOptimum) {
         if (isRPfromGroup) {
            // if router is a RP from group S than remove S from shared tree
            router.joinList.removeInterface(sourceAddress, destinationAddress, sourceInterface);
            // and send a pruneRPT to source S
            // re-send command by unicast
            sendToNeighbor(new PIMSMPruneRPTPacket(thisNode, pathSourceOptimum, TAMBYTESCMD,
                        sourceAddress, destinationAddress));
         } else {
            // forward command until RP router from group
            sendToNeighbor(new PIMSMPruneRPBitPacket(thisNode, pathRPOptimum, TAMBYTESCMD,
                         sourceAddress, destinationAddress));
         }
      }
      return;
   }
}



/**
  * Routine that take next hops to send data packets according to PIM/SM strategy
  * Verify if messages use: RPTtree or SPTtree
  */

public int[] getNextHops(Packet p_) {

   /*
    * Set enviroment variables from router
    */
   Packet packetReceived;
   int sourceInterface, thisNode, sourceAddress, destinationAddress, routeRPfromGroup;
   int kindOftree, kindOfsource;
   boolean isInterfacetoGroup, isRPfromGroup;

   // verify if router receive report/leave from host
   onExecute();

   // set local variables from packet

   sourceInterface = p_.getSource();
   thisNode = router.getNetworkId();

   sourceAddress = ((Packet)p_.getData()).getSource();
   destinationAddress = ((Packet)p_.getData()).getDestination();
   packetReceived = (Packet) (p_.getData());

   // verify if the router is RP from group
   routeRPfromGroup = router.groupRPList.getRendezVousPoint(destinationAddress);
   isRPfromGroup = false;
```

```
if (thisNode == routeRPfromGroup)
   isRPfromGroup = true;
// router is RP to destinationAddress
// verify if this router is interface to a group from host - DR from group
isInterfacetoGroup = false;      // initialize
// verify if is first-hop
if ((sourceAddress == sourceInterface) && !isRPfromGroup) {
   isInterfacetoGroup = true;
}
else {
   int[] neighborHosts = router.getNeighborHostsInGroup(destinationAddress);
   if (neighborHosts.length > 0) {
      isInterfacetoGroup = true;
   }
}
// take the type of transmission to the group
kindOfsource = router.kindOfSourceList.getValueFromSG(sourceAddress,
               destinationAddress);
if (kindOfsource == 0) {
   kindOfsource = router.kindOfSourceList.getValueFromSG(ANYSOURCE,
                  destinationAddress);
   if (kindOfsource == 0) {
      kindOfsource = REGISTER_ACTIVE;
      router.kindOfSourceList.setValueToSGOnList(ANYSOURCE,
      destinationAddress,REGISTER_ACTIVE);
   }
}

// take the treetype from receiver
kindOftree = router.kindOfTreeList.getValueFromSG(sourceAddress, destinationAddress);
if (kindOftree == 0) {
   kindOftree = router.kindOfTreeList.getValueFromSG(ANYSOURCE, destinationAddress);
   if (kindOftree == 0)
      kindOftree = RPTTREE;   // if it's not on list use default value
}

// take the unicast path to RP
int pathRPOptimum = 0;
// if router is not a RP to group
if (!isRPfromGroup)
   pathRPOptimum = (staticAlgorithm.getNextHops (new Packet(thisNode,
               routeRPfromGroup, UNICAST_TYPE, 0,
               new Packet(thisNode, routeRPfromGroup, UNICAST_TYPE, 0))))[0];

// take the unicast path to source S
int pathSourceOptimum = 0;
if (sourceAddress != ANYSOURCE)
   pathSourceOptimum = (staticAlgorithm.getNextHops (new Packet(thisNode,
                       sourceAddress, UNICAST_TYPE, 0,
                       new Packet(thisNode, sourceAddress,UNICAST_TYPE, 0))))[0];

// verify if DR: host is connect to router???
if (isInterfacetoGroup && !isRPfromGroup) {
   // if true: verify if is the first ou last hop from message
   // if first need to send message to tree
   if (sourceAddress == sourceInterface) {
      // first hop
      // verify if commands are sending in Register packets
      if (kindOfsource == REGISTER_ACTIVE) {
         // if REGISTER then send command to RP by unicast transmission
         // unicast transmission
         send(new PIMSMRegisterPacket(thisNode, routeRPfromGroup,
              TAMBYTESCMD, packetReceived,sourceAddress, destinationAddress),
              RouterPacket.MAX_TTL);
          return new int[0];
      } else {
         // if first hop and use tree verify if RPT or SPT
         int[] result = new int[1];
         if (kindOftree == RPTTREE) {
            // send data by RPF towards RP
            result[0] = pathRPOptimum;
         } else {
            // if SPT then send to all interfaces on the joinlist to S,G
```

```
                int numHosts = router.joinList.getInterfacesSize(sourceAddress,
                             destinationAddress);
                Vector destinations = router.joinList.getInterfaces(sourceAddress,
                                    destinationAddress);

                int[] nextHops = new int[0];
                if (numHosts > 0)
                    nextHops = new int[destinations.size()];
                    int nextItem = 0;

                    for (int i=0; i< numHosts;i++) {
                       // send the packet to destinationAddress
                       if (((Integer) destinations.get(i)).intValue() != sourceInterface) {
                           nextHops[nextItem] = ((Integer) destinations.get(i)).intValue();
                           nextItem++;
                       }
             }
             result = new int[0];
             if (numHosts > 0)
                result = new int[nextItem + destinations.size()];
             if (nextItem > 0) {
                 System.arraycopy(nextHops, 0, result, 0, nextItem);
          }

          if (numHosts > 0) {
             for (int i = 0; i < destinations.size(); i ++)
             result[nextItem+i] = ((Integer) destinations.get(i)).intValue();
          }
          return result;
       }
       return result;
    }
 } else {
    // else is the last hop: then is a receiver
    // in this case verify if use RPT or SPT tree
    if (kindOftree == RPTTREE) {
        // if is a RPT then do a control of received messages to group
             int count = router.messageCounterList.getNumMessagesFromSGroup(sourceAddress,
                             destinationAddress);
             if (count == 0) {
                // then add group to list
                router.messageCounterList.addSGroupOnList(sourceAddress,
                       destinationAddress, (int) 1);
             } else {
                // increment counter for group
                router.messageCounterList.incCountFromSGroupOnList(sourceAddress,
                       destinationAddress);
                if (router.messageCounterList.getNumMessagesFromSGroup (sourceAddress,
                    destinationAddress) > router.threshold) {
                   // this DR changes S,G to an SPTtree, because S is sending packets to G
                   // send joinSPT command towards Source
                   router.kindOfTreeList.setValueToSGOnList(sourceAddress,
                          destinationAddress, SPTTREE);
                   sendToNeighbor(new PIMSMJoinSPTPacket(thisNode, pathSourceOptimum,
                                TAMBYTESCMD, sourceAddress, destinationAddress));
                }  // else //  does not exceed threshold
             }
          }
       }
    } else {
       // if is not a DR then verify if router is a RP
         if (isRPfromGroup) {
           if (sourceAddress != sourceInterface) {
             // if true: router is RP from group
             if (sourceAddress != ANYSOURCE) {
                // verify if received a message from specific source
                if (kindOftree == RPTTREE) {
                   // in this case : then RP send a registerstop to sender
                   // unicast transmission
                   send(new PIMSMRegisterStopPacket(thisNode, sourceInterface,
                        TAMBYTESCMD, sourceAddress, destinationAddress), RouterPacket.MAX_TTL);
                }
             }
```

```
            // verify if RP has routers on joinlist to destinationAddress
            if (!(router.joinList.isInterfacePresent(ANYSOURCE, destinationAddress,0))) {
               sendToNeighbor(new PIMSMPrunePacket(thisNode,pathSourceOptimum,
                              TAMBYTESCMD, sourceAddress, destinationAddress));
               return new int[0];
            }
         }
      } // else is neither DR / RP
   }

   // forwarding the packets according to joinlist
   int numHosts = router.joinList.getInterfacesSize(ANYSOURCE, destinationAddress);
   Vector destinations = router.joinList.getInterfaces(ANYSOURCE, destinationAddress);

   int[] nextHops = new int[0];
   if (numHosts > 0)
      nextHops = new int[destinations.size()];
   int nextItem = 0;

   for (int i=0; i< numHosts;i++) {
      // send the packet to destinationAddress
      if (((Integer) destinations.get(i)).intValue() != sourceInterface) {
         nextHops[nextItem] = ((Integer) destinations.get(i)).intValue();
         nextItem++;
      }
   }

   int [] result = new int[0];
   if (numHosts > 0)
      result = new int[nextItem + destinations.size()];

   if (nextItem > 0) {
      System.arraycopy(nextHops, 0, result, 0, nextItem);
   }

   if (numHosts > 0) {
      for (int i = 0; i < destinations.size(); i ++)
         result[nextItem+i] = ((Integer)
      destinations.get(i)).intValue();
   }
   return result;
}

public void onTimer(Object o_) {

   int thisNode = router.getNetworkId();
   if (o_ == joinTimer) {
      // verify for all entrys if timeout on the joinlist

      Set set = router.joinList.keySet();
      Iterator iter = set.iterator();
      while ( iter.hasNext() ) {
         PIMInterfaceListKey ik = (PIMInterfaceListKey)( iter.next() );
         int timerSG = router.timerCounterList.getTimeFromSGroup(ik.source, ik.group);
         if (timerSG != TIME_INFINITO) {
              if (timerSG > router.joinDiscardInterval) {
               // remove entry from lists - timeout from entry
               router.joinList.removeAllInterfaces(ik.source, ik.group);
               router.timerCounterList.removeSGroupFromList(ik.source, ik.group);
            }
         }
      }
      // restart timer
      setTimer(router.joinDiscardInterval, joinTimer);
  } else {
      // once a minute: send a join to RP (if ANYSOURCE,G) or to source S (if S,G)
      int path, rp;
      // verify for all entrys if timeout on the joinlist
      Set set = router.joinList.keySet();
      Iterator iter = set.iterator();
      while ( iter.hasNext() ) {
         PIMInterfaceListKey ik = (PIMInterfaceListKey)( iter.next() );
         if (ik.source == ANYSOURCE) {
```

```
                // take the unicast path to RP
                rp = router.groupRPList.getRendezVousPoint(ik.group);
                if (thisNode == rp)
                   break;
                path = (staticAlgorithm.getNextHops (new Packet(thisNode, rp,
                         UNICAST_TYPE, 0, new Packet(thisNode, rp,
                         UNICAST_TYPE, 0))))[0];
             } else {
                // take the unicast path to source S
                path = (staticAlgorithm.getNextHops (new Packet(thisNode,
                            ik.source, UNICAST_TYPE, 0,
                            new Packet(thisNode, ik.source, UNICAST_TYPE, 0))))[0];
             }
             sendToNeighbor(new PIMSMJoinPacket(router.getNetworkId(), path,
                          TAMBYTESCMD, ik.source, ik.group));
          }
       }
       // restart timer from refresh once a min.
       setTimer(60 , refreshTimer);
    }
}
```

A.3.3 PIMSMJoinPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a Join command from PIMSM
 */

public class PIMSMJoinPacket extends PIMPacket {

   public static final PacketType PACKET_TYPE = new PacketType("PIMSM_JOIN");

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */

   public PIMSMJoinPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
      super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
   }
}
```

A.3.4 PIMSMJoinRPTPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a JoinRPT command from PIMSM
 */

public class PIMSMJoinRPTPacket extends PIMPacket {

   public static final PacketType PACKET_TYPE = new PacketType("PIMSM_JOINRPT");

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */

   public PIMSMJoinRPTPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
      super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
   }
}
```

A.3.5 PIMSMJoinSPTPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a Join SPT command from PIMSM
 */

public class PIMSMJoinSPTPacket extends PIMPacket {

   public static final PacketType PACKET_TYPE = new PacketType("PIMSM_JOINSPT");

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */

   public PIMSMJoinSPTPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
      super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
   }
}
```

A.3.6 PIMSMPrunePacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a Prune command from PIMSM
 */

public class PIMSMPrunePacket extends PIMPacket {

   public static final PacketType PACKET_TYPE = new PacketType("PIMSM_PRUNE");

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */

   public PIMSMPrunePacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
      super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
   }
}
```

A.3.7 PIMSMPruneRPTPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a PruneRPT command from PIMSM
 */

public class PIMSMPruneRPTPacket extends PIMPacket {

   public static final PacketType PACKET_TYPE = new PacketType("PIMSM_PRUNERPT");

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */

   public PIMSMPruneRPTPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
      super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
   }
}
```

A.3.8 PIMSMPruneSPTPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a PruneRP command from PIMSM
 */

public class PIMSMPruneSPTPacket extends PIMPacket {

   public static final PacketType PACKET_TYPE = new PacketType("PIMSM_PRUNESPT");

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */

   public PIMSMPruneSPTPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
      super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
   }
}
```

A.3.9 PIMSMPruneRPBitPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a PruneRPBit command from PIMSM
 */

public class PIMSMPruneRPBitPacket extends PIMPacket {

public static final PacketType PACKET_TYPE = new PacketType("PIMSM_PRUNERPBIT");

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */

   public PIMSMPruneRPBitPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
      super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
   }
}
```

A.3.10 PIMSMRegisterPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a Register command from PIMSM
 */

public class PIMSMRegisterPacket extends PIMPacket {

   public static final PacketType PACKET_TYPE = new PacketType("PIMSM_REGISTER");

  /**
    * Generic handle for holding data.
    */

   /**
    * Constructs a new packet with the specified characteristics.
    * No consistencies are made on the validity of the entered data.
    */
```

```
    public PIMSMRegisterPacket(int from_, int to_, int size_, Packet data_,
                               int fromHost_, int toHost_) {
       super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
       data = data_;
    }

    /**
     * Returns the object held by this packet.
     */

    public Object getData()     { return data; }
}
```

A.3.11 PIMSMRegisterStopPacket.java

```
package simmcast.route;

import simmcast.network.*;

/**
 * This class identify a Register command from PIMSM
 */

public class PIMSMRegisterStopPacket extends PIMPacket {

    public static final PacketType PACKET_TYPE = new PacketType("PIMSM_REGISTERSTOP");

    /**
     * Constructs a new packet with the specified characteristics.
     * No consistencies are made on the validity of the entered data.
     */

    public PIMSMRegisterStopPacket(int from_, int to_, int size_, int fromHost_, int toHost_) {
       super(from_, to_, PACKET_TYPE, size_, fromHost_, toHost_);
    }
}
```

A.3.12 PIMSMList.java

```
package simmcast.route;

import java.util.*;
import java.io.*;


/**
 * This class implements a list of values.
 */

public class PIMSMList {

    private Hashtable table = new Hashtable();

    // get value on list
    public int getValueFromSG(int source_, int group_) {
       Integer i = (Integer)(table.get(new PIMInterfaceListKey(source_, group_)) );
       return i == null ? 0 : i.intValue();
    }

    // add new group on list
    public void addSGOnList(int source_, int group_, int value_) {

       PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
       if (table.get(key) != null)
          table.remove(key);
       table.put(key, new Integer(value_));
    }

    // remove group from list
    public void removeSGFromList(int source_, int group_) {
       PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
```

```
      table.remove(key);
   }

   public void setValueToSGOnList(int source_, int group_, int value_) {
      addSGOnList(source_, group_, value_);
   }
}
```

A.3.13 PIMSMMessagesCounterList.java

```
package simmcast.route;

import java.util.*;
import java.io.*;


/**
 * This class implements a list of trees.
 */

public class PIMSMMessagesCounterList {

   private Hashtable table = new Hashtable();

   // Verify how many messages was received by sourcer by group
   public int getNumMessagesFromSGroup(int source_, int group_) {
      int i = 0;
      Vector list = (Vector) table.get(new PIMInterfaceListKey(source_, group_));
      return list == null ? 0 : ((Integer) list.get(i)).intValue();
   }

   // Add source, group on list
   public void addSGroupOnList(int source_, int group_, int numMessages_) {

      PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
      Vector list = (Vector)( table.get(key) );
      if (list == null) {
         list = new Vector();
         table.put(key, list);
      }
      Integer i = new Integer(numMessages_);
      if (list.indexOf(i) == -1)
         list.addElement(i);
   }

   // Remove source, group from list
   public void removeSGroupFromList(int source_, int group_) {
      PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
      table.remove(key);
   }

   // Increase num messages received from source, group
   public void incCountFromSGroupOnList(int source_, int group_) {

      PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
      Vector list = (Vector)( table.get(key) );
      int count = 1, i = 0;        // count at least size 1
      if (list != null) {
         count = ((Integer) list.get(i)).intValue() + 1;
         removeSGroupFromList(source_, group_);
      }
      addSGroupOnList(source_, group_, count);
   }
}
```

A.3.14 PIMSMTimeGroupList.java

```
package simmcast.route;

import java.util.*;
import java.io.*;
```

```
/**
 * This class implements a list of trees.
 */

public class PIMSMTimeGroupList {

   private Hashtable table = new Hashtable();

   // Verify how many messages was received by sourcer by group
   public int getTimeFromSGroup(int source_, int group_) {
      int i = 0;
      Vector list = (Vector) table.get(new PIMInterfaceListKey(source_, group_));
      return list == null ? 0 : ((Integer) list.get(i)).intValue();
   }

   // Add source, group on list
   public void addSGroupOnList(int source_, int group_, int time_) {
      PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
      Vector list = (Vector)( table.get(key) );
      if (list == null) {
         list = new Vector();
         table.put(key, list);
      }
      Integer i = new Integer(time_);
      if (list.indexOf(i) == -1)
         list.addElement(i);
   }

   // Remove source, group from list
   public void removeSGroupFromList(int source_, int group_) {
      PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
      table.remove(key);
   }

   // Increase num messages received from source, group
   public void setTimeFromSGroupOnList(int source_, int group_, int time_) {

      PIMInterfaceListKey key = new PIMInterfaceListKey(source_, group_);
      Vector list = (Vector)( table.get(key) );
      if (list != null)
         list.removeElement(key);
      addSGroupOnList(source_, group_, time_);
   }
}
```

A.3.15 PIMSMRendezVousPointList.java

```
package simmcast.route;

import java.util.*;
import java.io.*;


/**
 * This class implements a list of rendezvous points.
 */

public class PIMSMRendezVousPointList {

   private Hashtable table = new Hashtable();

   // take associated RP from group
   public int getRendezVousPoint(int group_) {
      Integer rp = (Integer) table.get(new Integer(group_));
      if (rp == null)
         return 0;
      return rp.intValue();
   }
```

```
    public void addRendezVousPoint(int group_, int nodeRP_) {
       table.put(new Integer(group_), new Integer(nodeRP_));
    }
}
```

Bibliografia

[1] K. C. Almeroth , "The Evolution of Multicast: From MBone to Interdomain Multicast to Internet2 Deployment", IEEE Network, January/February 2000.

[2] A. Ballardie, P. Francis & J. Crowcroft, "Core Based Tree (CBT): An Architecture for Scalable Multicast Routing", In Proceedings of the ACM SIGCOMM, San Francisco, 1993.

[3] A. Ballardie, P. Francis & J. Crowcroft, "Core Based Tree (CBT version 2) Multicast Routing: Protocol Specifications", RFC2189, IETF, September 1997, `http://www.ietf.org/rfc/rfc2189.txt`.

[4] J. Banks, "Handbook of Simulation: Principles, Metodology, Advances, Applications and Practice", John Wisley & Sons, New York, 1998.

[5] M. Barcellos & V. Roesler, "M&M:Multicast & Multimídia", Em SBC´2000, XX Congresso da Sociedade Brasileira de Computação, SBC 2000.

[6] M. Barcellos, H. Muhammad & A. Detsch, "Simmcast: a Simulation Tool for Multicast Protocol Evaluation", XIX Brazilian Symposium on Computer Networks (SBRC 2001), Proceedings, SBC, Florianópolis, 21-25 May 2001.

[7] T. Bates, R. Chandra, D.Katz & Y. Rekhter, "Multiprotocol Extensions for BGP-4", RFC2283, IETF, February 1998, `http://www.ietf.org/rfc/rfc2283.txt`.

[8] G. M. Birtwistle, O-J. Dahl, B. Myhrhaung & K. Nygaard, "Simula Begin", Academic Press, 1973.

[9] L. S. Brakmo & L. L. Peterson, "Experiences with Network Simulation", In ACM Sigmetrics'96, May 1996.

[10] L. Breslau et alli, "Advances in Network Simulation", IEEE Computer, May 2000.

[11] L. Breslau, S. Bajaj, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kummar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, H. Yu, Y. Xu & D. Zappala, "Improving Simulation for Network Research", IEEE Computer Magazine, September 1999.

[12] L. H. Costa, S. Fdida & O. Duarte, "Hop by Hop Multicast Routing Protocol", ACM SIGCOMM 2001, **Proceedings**, San Diego, California, USA, Aug. 2001.

[13] Y. K. Dalal & R. M. Metcalfe, "Reverse Path Forwarding of Broadcast Packets", Communications of the ACM, December 1998.

[14] S. Deering, "Host Extension for IP Multicasting", RFC1112, IETF, August, 1989, `http://www.ietf.org/rfc/rfc1112.txt`.

[15] S. Deering, D. Estrin, D. Farinacci, M. Handley, A. Helmy, V. Jacobson, C. Liu, P. Sharma, D. Thaler & L. Wei, "Protocol Independent Multicast - Sparse Mode (PIM-SM): Motivation and Architecture", Proposed Experimental RFC, September 1996.

[16] D. Estrin, M. Handley, A. Helmy, P. Huang & David Thaler, "A Dynamic Bootstrap Mechanism for Rendezvous-based Multicast Routing Deborah Estrin", In Proceedings of the IEEE INFOCOM, 1999.

[17] D. Estrin, A. Helmy, V. Jacobson & L. Wei, "Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification", Proposed Experimental RFC, September, 1996.

[18] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma & L. Wei, "Protocol Independent Multicast Sparse Mode (PIM-SM): Protocol Specification", RFC2362, IETF, June 1998, `http://www.ietf.org/rfc/rfc2362.txt`.

[19] D. Estrin, M. Handley, J. Heidemann, S. McCanne, Y. XSu & H. Yu, "Network Visualization with the VINT Network Animator Nam", IEEE Computer Magazine, November 1999.

[20] K. Fall & HK. Varadhan, "The ns Manual", The VINT Project, UC Berkeley, LBL,USC/ISI, Xerox PARC, `http://www.isi.edu/nsman/ns/ns-documentation.html`.

[21] R. Fenner, "Internet Group Management Protocol: version 2", RFC2236, IETF, November 1997, `http://www.ietf.org/rfc/rfc2236.txt`.

[22] E. Gamma et alli, "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[23] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley, 2000.

[24] Georgia Tech Internetwork Topology Models (gt-itm), `http://www.cc.atech.edu//projects/gtimss`.

[25] R. Govindan & A. Reddy, "An Analysis of Internet Inter-Domain Topology and Route Stability", In Proceedings of the IEEE INFOCOM, 1997.

[26] P. Huang, D. Estrin & J. Heidemannn, "Enabling Large-Scale Simulations: Selective Abstraction Approach to the Study of Multicast Protocols", In Proceedings of the Int'l Symp. Modeling, Analasys and Simulation of Computer and Telecommunication System, IEEE CS Press, California, 1998.

[27] C. Huitema, "Routing in the Internet", Englewood Cliffs, NJ: Prentice Hall, 1995.

[28] H. Huni, R. Johnson & R. Engel, "A Framework for Network Protocol Software". In Conference on Object-Oriented Programming: System, Languages and Applications (OPSPLA´95), Austin, TX, September 1995.

[29] D. B. Ingham & G. D. Parrington, "Delayline: A Wide-Area Network Emulation Tool", Computing Systems, v.7, n.3, 1994.

[30] R. Jain, "The art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling", John Wisley & Sons, New York, 1991.

[31] R. E. Johnson, "Frameworks = (components+patterns)", Communications of the ACM, October, 1997.

[32] I. Keidar, R. Khazan, N. Lynch & A. Shvartsman, "An Inheritance-Based Techique for Building Simulation Proofs Incrementally", ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 1, January 2002.

[33] M. Kodialam & S. Low, "Resource Allocation in Multicast Trees Murali Kodialam", In Proceedings of the IEEE INFOCOM, 1999.

[34] B. Krupczak, K. Calvert & M. Ammar, "Implementing Protocols in Java: The Price of Portability", IEEE INFOCOM´98, San Francisco, 29 March-2nd, April 1998.

[35] S. Kumar, C. Alaettinoglu, P. Radoslavov, D. Estrin, M. Handley & D. Thaler, "The MASC/BGMP Architecture for Inter-domain Multicast Routing", In Proceedings of the ACM SIGCOMM'98, Vancouver, Canada, 1998.

[36] J. F. Kurose & K. W. Ross, "Computer Networking: A Top Down Approach Featuring The Internet", Boston: Addison-Wesley, 2000.

[37] A. M. Law & W. D. Kelton, "Simulation Modeling & Analysis", McGraw-Hill. NewYork. Second Edition, 1991.

[38] M. C. Little, "JavaSim Users Guide", Public Release 0.3, Version 1.0, `http://javasim.ncl.ac.uk`

[39] N. Lynch, "Distributed Algorithms", Morgan Kaufmann, San Francisco, 1996.

[40] T. Maufer, "Deploying IP Multicast in the Enterprise". Prentice-Hall. New Jersey. 1998.

[41] D. Madhava Rao, R. Radhakrishnan & P. A. Wilsey, "FWNS: A Framework for Web-Based Network Simulation", In Procceding of International Conference On Web-Based Modelling & Simulation, WEBSIM´99, Vol. 31, No. 3, January 1999.

[42] C. K. Miller, "Multicast Networking and Applications", Massachusetts: Addison-Wesley, 1999.

[43] J. Moy, "Multicast Extension to OSPF", RFC1584, March 1994, `http://www.ietf.org/rfc/rfc1584.txt`.

[44] H. H. Muhammad & M. Barcellos, "Simulating Group Communication Protocols Through an Object -Oriented Framework", 35th Annual Simulation Symposium (SS2002), Proceedings, IEEE (New York), San Diego, 14-18 April 2002.

[45] H. H. Muhammad, M. Barcellos & R. Casais, "Simulação de Roteamento na Avaliação de Protocolos Multicast e Sistemas Distribuídos de Grupo", I Workshop em Desempenho de Sistemas Computacionais e de Comunicação (Wperformance 2002), parte do XXII Congresso da SBC 2002, Anais, SBC (Porto Alegre), Florianópolis, 18-19 July 2002

[46] C. A. Noronha Jr. & F. A. Tabagi, "Optimum Routing of Multicast Streams", In Proceedings of the IEEE INFOCOM, 1994.

[47] J. K. Ousterhout, "Tcl and the Tk Toolkit", Addison-Wesley, Reading, MA, 1994.

[48] S. Paul, "Multicasting on the Internet and Its Applications", Boston: Kluwer Academic Publishers, 1998.

[49] G. D. Parrington et al, "The Design and Implementation of Arjuna". Broadcast Project Technical Report, October 19994.

[50] PARSEC, `http://pcl.cs.ucla.edu/projects/parsec`

[51] G. Phillips & S. Shenker, "Scaling of Multicast Trees". In Proceedings of the ACM SIGCOMM, September 1999.

[52] P. Radoslavov, C. Papadopoulos, R. Govindan & D. Estrin, "A Comparison of Application-level and Router-assisted Hierarchical Schemes for Reliable Multicast", INFOCOM 2001, Procedings, Alaska, 26-29 April 2001. IEEE (New York), April 2001.

[53] Y. Rekhter, T. J. Watson & T. Li, "A Border Gateway Protocol 4 (BGP-4)", RFC1771, IETF, March 1995, `http://www.ietf.org/rfc/rfc1771.txt`

[54] G Riley, M. Ammar & R. Fujimoto, "Stateless Routing in Network Simulations", Eighth International Symposium on Modeling, Analysis and Simulation of Computer and Telecom munication Systems (Mascots 2000), Proceedings, August 2000, San Francisco, CA.

[55] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols", ACM Computer Communication Review, January, 1997.

[56] L. Sahasrabuddhe & B. Mukherjee, "Multicast Routing: Algorithms and Protocols: A Tutorial", IEEE Network, January/February 2000.

[57] D. C. Schmidth, "The ADAPTATIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software", In 12th Annual Sun Users Group Conference, San Francisco, CA, 1994.

[58] SIMMCAST, `http://www.inf.unisinos.br/~simmcast`

[59] D. Thaler, D. Estrin & D. Meyer, "Border Gateway Multicast Protocol (BGMP): Protocol Specification", November 2000, `http://search.ietf.org/internet-drafts/draft-ietf-bgmp-spec-02.txt`.

[60] S. Ueno, T. Kato & K. Suzuki, "Analysis of Internet Multicast Traffic Performance Considering Multicast Routing Protocol", In Proceeding of the ICNP Conf, 2000.

[61] K. Varandhan, D. Estrin & S. Floyd, "Impact of Network Dynamics on End-to-End Protocols: Case Studies in Reliable Multicast", IEEE Symposium on Computers and Communications, July, 1998.

[62] S. Waitzman, S. Deering & C. Partridge, "Distance Vector Multicast Routing Protocol", RFC1075, Novermber, 1998, `http://www.ietf.org/rfc/rfc1075.txt`.

[63] T. Wang & R. Katz, "An Analysis of Multicast Forward State Scalability", In Proceeding of the ICNP Conf, 2000.

[64] L. Wei & D. Estrin, "The Trade-offs of Multicast Trees and Algorithms", In Proceedings of the International Conference on Computer Communications and Networks (ICCCN), 1994.

[65] B. Williamson, "Developing IP Multicast Networks", Volume I, Indianapolis: Cisco Press, 2000.

[66] R. Wittmann & T. Zitterbart, "Multicast Communication: Protocols and Applications", San Diego: Academic Press, 2001.