Universidade Federal Fluminense

Transmissão Multicast Confiável: a Implementação no TANGRAM II Whiteboard e a Análise de Desempenho

Tese submetida para a obtenção do título de

Mestre em Ciência da Computação

ao Programa de Pós-Graduação de

Engenharia de Ciência da Computação da UFF

por

Milena Scanferla Araujo

Dezembro 2002

Transmissão Multicast Confiável:
a Implementação no TANGRAM II Whiteboard
e a Análise de Desempenho

Milena Scanferla Araujo

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DE CIÊNCIA DA COMPUTAÇÃO DA UNIVERSIDADE FEDERAL FLUMINENSE COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Profa. Anna Dolejsi dos Santos, Dra.

Prof. Edmundo Albuquerque de Souza e Silva, Ph.D.

Prof. José Augusto Suruagy Monteiro, Ph.D.

Profa. Rosa Maria Meri Leao, Dra.

Profa. Noemi da La Rocque Rodriguez, Dra.

NITERÓI, RJ - BRASIL

DEZEMBRO DE 2002

Dedico este trabalho aos meus queridos pais,

Kalgen e Eliete, e ao meu noivo Marcelo.

Agradecimentos

Primeiramente agradeço a Deus, aos meus pais e ao meu noivo por terem estado ao meu lado durante toda esta jornada, sempre me dando força para continuar.

Agradeço a minha orientadora, professora Anna Dolejsi, que abriu as portas para esta caminhada tão importante na minha vida. Agradeço pela oportunidade que me ofereceu e pela confiança que depositou em mim.

Agradeço ao professor Edmundo, que sempre confiou no meu trabalho e me aceitou em sua equipe como sua aluna, além de ter me mostrado o quanto eu sou capaz. Não tenho palavras para descrever a minha admiração por ele!

Uma vez, recebemos um texto do professor Edmundo que tratava do relacionamento entre aluno e professor. E agora eu gostaria de dedicar este mesmo texto a ele:

"Não me ensine nada que eu possa descobrir,
provoque minha curiosidade.
Não me dê apenas respostas.
Desarrume minhas idéias e me dê
somente pistas de como ordená-las.
Não me mostre exemplos.

Antes me encoraje a ser exemplo
vivo de tudo que posso aprender.
Construa comigo o conhecimento.
Sejamos juntos inventores,
descobridores, navegadores e piratas
de nossa aprendizagem.
Não fale apenas de um passado distante
ou de um futuro imprescindível.
Esteja comigo hoje, alternando as
sensações de quem ensina e de quem aprende."

Agradeço às minhas amigas que tornaram a dura jornada diária de volta para casa uma viagem divertida. E, especialmente, a Carolina que teve toda a paciência, me ajudando a revisar o texto desta dissertação.

Agradeço aos alunos do laboratório LAND, que tiveram paciência em me ajudar sempre que precisei. Em especial, ao Allyson que trabalhou comigo todo este tempo, dividindo problemas e conquistas. E ao Sadoc, que teve um papel extremamente importante neste trabalho de dissertação.

Agradeço à CAPES, pela bolsa de estudos recebida e aos projetos COMIT e MAPPED do CNPq pelo suporte proporcionado, tornando possível o desenvolvimento deste trabalho.

Enfim, gostaria de agradecer a todos que estiveram envolvidos direta ou indiretamente no desenvolvimento e conclusão de minha dissertação.

Muito obrigada a todos!

Resumo da Tese apresentada à UFF como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciência da Computação (M.Sc.)

Transmissão Multicast Confiável: a Implementação no TANGRAM II Whiteboard e a Análise de Desempenho

Milena Scanferla Araujo

Dezembro/2002

Orientador: Anna Dolejsi dos Santos
Programa: Engenharia de Ciência da Computação

O objeto de estudo desta dissertação concentra-se no desenvolvimento e na análise da confiabilidade na transmissão Multicast.

Para a garantia da confiabilidade na transmissão Multicast foi desenvolvida a biblioteca *Reliable Multicast Library* (RML), com a função de detectar perdas de pacotes e de recuperar eficientemente os pacotes perdidos, tornando confiável a transmissão. Ela pode ser utilizada por diversos tipos de aplicações e oferece rotinas para a garantia da confiabilidade na entrega dos dados em um ambiente Multicast não confiável.

Além da implementação da biblioteca Multicast confiável, foi desenvolvido um modelo que simula o seu comportamento. Este comportamento pode ser analisado através dos resultados obtidos da simulação do modelo. O modelo descreve o comportamento do algoritmo de transmissão Multicast confiável diante das variações de retardo e do processo de perdas encontradas em uma rede de transmissão de dados.

Como resultados deste trabalho estão disponibilizados, em domínio público, a biblioteca RML, que garante a confiabilidade em uma transmissão Multicast, e o modelo que a descreve, tornando possível a análise de desempenho e o ajuste de parâmetros para diferentes condições da Internet. Além disso, a distribuição internacional do TANGRAM II Whiteboard.

Abstract of Thesis presented to UFF as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

Reliable Multicast Transmission: the Implementation in the TANGRAM II Whiteboard and its Performance Evaluation

Milena Scanferla Araujo

December/2002

Advisor: Anna Dolejsi dos Santos
Department: Computer Science Engineering

The dissertation focuses on the analysis and development of a reliable Multicast transmission.

In order to guarantee reliability in Multicast transmissions, a Reliable Multicast Library (RML) that can detect and recover lost packages efficiently was developed. This library consists of a set of routines that guarantee data reliability in an unreliable Multicast network and can be used by different applications.

Besides the implementation of a Reliable Multicast Library, a simulation model was developed to analyze the performance of the reliable Multicast algorithm implemented. The model is useful to study the behavior of the reliable Multicast transmission algorithm under a variety of scenarios which includes different loss rates and network delay found in the Internet.

Among the major results from this work are the implementation of the RML, which is available to download under the GNU free-license, and its model. Thus it is possible to evaluate the performance of the RML algorithm under different parameter values and tune these values according to network conditions. In addition, the international distribution of TANGRAM II Whiteboard.

Glossário

Grupo Multicast : formado pelo conjunto de *hosts* de uma rede, onde dados são compartilhados via conexão Multicast. Esses grupos são identificados por um endereço Multicast. Cada emissor envia pacotes para um endereço de grupo (por exemplo: 233.7.124.1), no qual estarão associados diversos receptores;

Membro Multicast : *hosts* participantes de um grupo Multicast.

Sessão Multicast : representa o tempo em que um determinado grupo Multicast está ativo. Basta que o grupo tenha um único membro para o considerarmos um grupo ativo;

ACK : pacote especial usado para confirmação do recebimento de dados;

NACK : pacote especial usado para informar a perda de dados;

Cache : estrutura de dados mantida pelos membros do grupo Multicast, com a função de armazenar informações importantes sobre o grupo;

Temporizador : intervalo de tempo que um usuário espera para executar uma determinada ação (evento);

Lista de Eventos : lista com os eventos escalonados para serem executados em algum tempo no futuro.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

ESTE trabalho de dissertação abrange a implementação da confiabilidade na transmissão Multicast, através da biblioteca RML (*Reliable Multicast Library*), a criação do modelo que a descreve e a análise de seu desempenho através do modelo criado.

Nas seções seguintes são apresentadas: (1.1) a motivação e o objetivo que introduzem todo o trabalho realizado; (1.2) a contribuição deste trabalho de dissertação; e (1.3) o roteiro que resume a dissertação.

1.1 Motivação e Objetivo

Com o advento de aplicações multimídia distribuídas, torna-se crescente a necessidade de processamento e de largura de banda suficientes para a distribuição desses tipos de dados. Por isso, se faz necessária uma distribuição de dados eficiente, fazendo com que a utilização da rede seja mais eficaz.

Com a transmissão Multicast é possível reduzir o número de pacotes duplicados no canal, economizando largura de banda e, consequentemente, otimizando o uso da rede. Como o Multicast utiliza o protocolo UDP para transmissão, cada pacote pode trafegar por um caminho diferente do emissor até o receptor. Portanto, além dos pacotes poderem se perder durante a transmissão, eles ainda podem chegar fora de ordem, tornando o protocolo Multicast não confiável. Para a garantia da confiabilidade na transmissão Multicast, foi desenvolvida a biblioteca RML, com a função de detectar perdas de pacotes e recuperar eficientemente os pacotes perdidos, tornando a transmissão confiável. Ela pode ser utilizada por diversos tipos de aplicações e oferece rotinas para a garantia da confiabilidade na entrega dos dados em um ambiente Multicast.

O objeto de estudo desta dissertação concentra-se no desenvolvimento e na análise da confiabilidade na transmissão Multicast. A biblioteca desenvolvida foi minuciosamente testada através de experimentos na Internet, confirmando a sua robustez e confiabilidade na transmissão Multicast.

Como a modelagem de sistemas de computação e comunicação é uma tarefa essencial no processo de desenvolvimento de novas tecnologias, além da implementação da biblioteca Multicast confiável, foi desenvolvido um modelo que simula o seu comportamento. Este comportamento pode ser analisado através dos resultados obtidos da simulação do modelo. Considerando que nem sempre é possível utilizar os recursos da rede para a análise de desempenho da transmissão Multicast confiável, o modelo descreve o comportamento do algoritmo de transmissão Multicast confiável diante das variações de retardo e do processo de perdas encontradas em uma rede de transmissão de dados.

1.2 Contribuição

Criou-se uma biblioteca para garantir a confiabilidade na transmissão Multicast, que pode ser usada por diversas aplicações distribuídas. Esta biblioteca foi implementada em conjunto com dois integrantes do laboratório de modelagem, análise e desenvolvimento de redes e computação de sistemas (LAND), da Universidade Federal do Rio de Janeiro (UFRJ)[4]. O trabalho em equipe também abrange a integração desta biblioteca com a ferramenta *whiteboard* TGWB para garantia da confiabilidade na transmissão dos dados enviados e recebidos pela ferramenta.

A contribuição inclui a elaboração do modelo Multicast confiável, com a finalidade de analisar o desempenho da biblioteca diante das diferentes situações que a rede pode apresentar. Através do modelo, é possível analisar o comportamento de acordo com a variação do atraso na rede, das perdas de dados na transmissão e do intervalo dos temporizadores.

1.3 Roteiro

O Capítulo 2 descreve os diferentes tipos de protocolos para transmissão Multicast confiável. O Capítulo 3 apresenta a ferramenta TANGRAM II Whiteboard, uma aplicação distribuída que utiliza a biblioteca RML para garantia da confiabilidade na transmissão Multicast. Nos Capítulos 4 e 5 são descritas as implementações da biblioteca RML e do modelo que a descreve, respectivamente. A análise de desempenho da biblioteca através do modelo e os resultados obtidos são apresentados no Capítulo 6. E no Capítulo 7 são apresentados as conclusões e os possíveis trabalhos futuros.

Capítulo 2

Protocolos para Transmissão Multicast Confiável

ESTE capítulo tem o objetivo de apresentar as características de uma transmissão Multicast, e os mecanismos que a tornam confiável, fazendo uma análise do desempenho de cada um destes mecanismos.

2.1 Introdução

Nos últimos anos, com o advento de aplicações multimídia distribuídas tais como transmissões de imagens, texto, áudio e vídeo na Internet, torna-se crescente a necessidade de processamento e de largura de banda suficientes para a distribuição desses tipos de dados. Apesar do grande avanço tecnológico alcançado hoje, nem todos possuem acesso aos meios de transmissão adequados a essas aplicações. Por isso, se faz necessária uma distribuição de dados eficiente, fazendo com que a utilização da rede seja mais eficaz.

Existem algumas maneiras de economizar banda de transmissão. Dependendo da aplicação, a compactação dos dados antes do envio pode reduzir muito a banda necessária para a transmissão. Em outros casos, um mesmo conjunto de dados é enviado para muitos nós da rede ao mesmo tempo, através de várias transmissões independentes (uniponto ou Unicast). Nesta situação, no mesmo canal podem estar passando várias cópias de um mesmo pacote. Este tipo de transmissão pode ser otimizada, reduzindo o número de pacotes duplicados no canal com o uso da transmissão multiponto, ou transmissão Multicast.

O Multicast é voltado para aplicações do tipo um-para-muitos, onde somente um *host* envia os dados para os demais e muitos-para-muitos, onde todos os *hosts* podem enviar e receber dados simultaneamente. Nestes casos, o Multicast apresenta vantagens quando comparado com os mecanismos de transmissão Unicast. Neste caso, se o *host* emissor deseja enviar dados para n *hosts* receptores, então n conexões e n cópias serão feitas para cada pacote de dados a ser enviado, fazendo uma péssima utilização da rede. A transmissão Multicast também é mais eficiente do que a Broadcast, pois este último envia os dados para toda a rede de forma indiscriminada. Isto também resulta em desperdício de recursos, pois implica em transportar os dados para todos os *hosts* da rede, mesmo que o número de receptores desejosos daqueles dados seja reduzido. Com o Multicast, o emissor envia apenas uma única cópia dos pacotes pelo canal, para um endereço de grupo Multicast. Os roteadores da rede replicam estes pacotes de forma inteligente, encaminhando os dados de acordo com a localização dos receptores interessados naquela informação.

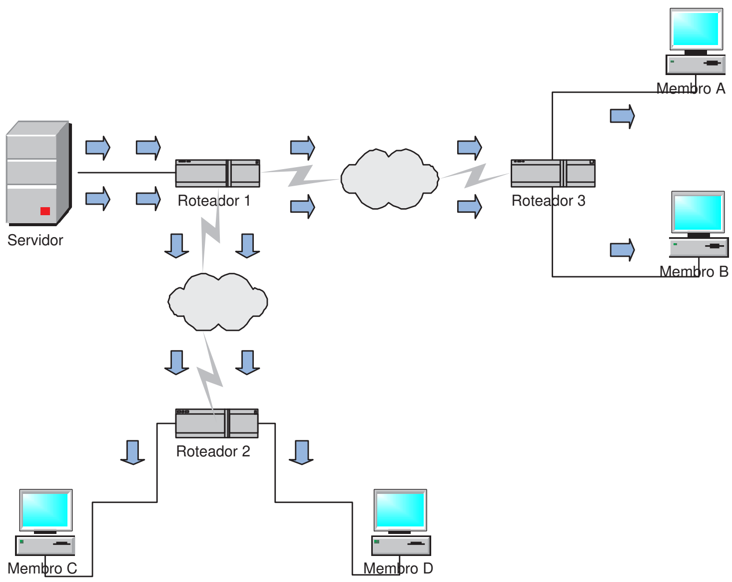

Figura 2.1: Transmissão Unicast

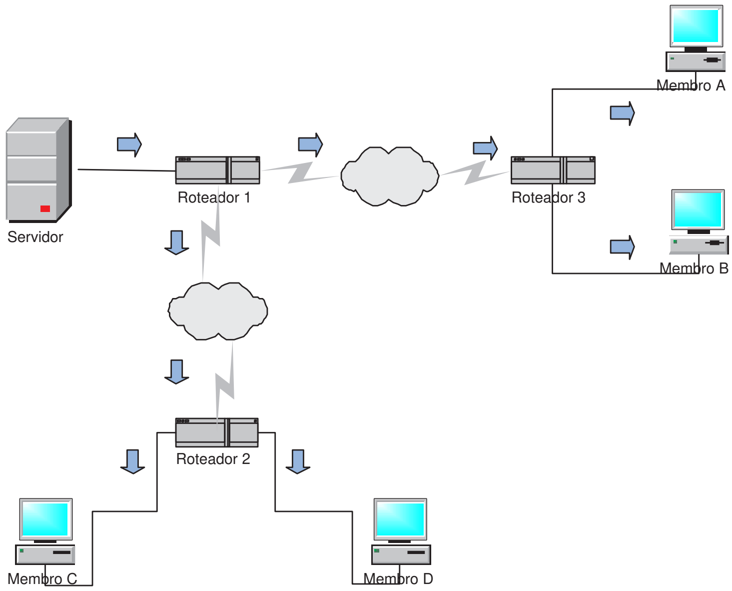

Figura 2.2: Transmissão Multicast

As Figuras 2.1 e 2.2 exemplificam transmissões via conexão Unicast e Multicast, respectivamente. De acordo com a Figura 2.1, quando o servidor deseja transferir um pacote da dados para os quatro membros da rede, ele faz uma cópia do pacote para que cada um dos membros, aumentando o número de pacotes que trafegam em cada canal da rede. Cada pacote de dados vai ser enviado em uma conexão Unicast diferente. A Figura 2.2 mostra que para enviar o mesmo pacotes de dados para destinos diferentes, não é necessária a duplicação de pacotes no canal. Cada roteador fica encarregado de fazer cópias dos pacotes de dados e de enviar apenas um pacote de dados por cada porta do roteador.

O Multicast pode ser extremamente útil em diferentes situações, como as listadas a seguir:

- um grupo de usuários querendo fazer a atualização de um determinado software;
- uma vídeo conferência;
- atualizações de banco de dados distribuídos;
- simulações distribuídas;

- jogo com múltiplos jogadores.

O intervalo de endereços IP é dividido em classes, onde os endereços IP Multicast estão na faixa entre 224.0.0.0 e 239.255.255.255, como ilustrado na Figura 2.3.

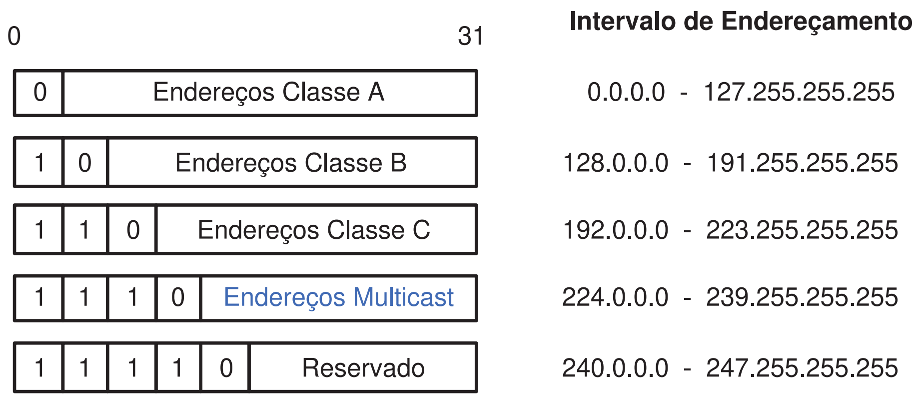

Figura 2.3: Endereçamento IP

Todo pacote direcionado a um endereço IP que comece com os bits 1110 é um pacote Multicast. Os outros 28 bits identificam o grupo Multicast para o qual este pacote está sendo direcionado.

A Tabela 2.1 mostra alguns endereços Multicast que são reservados para usos especiais [2].

Endereço	Função
224.0.0.1	Todos os *hosts* em uma LAN
224.0.0.2	Todos os roteadores em uma LAN
224.0.0.4	Todos os roteadores DVMRP em uma LAN
224.0.0.5	Todos os roteadores OSPF em uma LAN
224.0.0.6	Todos os roteadores OSPF designados em uma LAN
224.0.0.13	Todos os roteadores PIM em uma LAN

Tabela 2.1: Endereços especiais Multicast

O intervalo de 224.0.0.0 até 224.0.0.255 é reservado para propósitos locais (tarefas administrativas e de manutenção), sendo que os pacotes destinados a eles nunca atravessam o roteador Multicast. Igualmente, o intervalo de 239.0.0.0 até 239.255.255.255 está reservado para tarefas administrativas em geral, não necessariamente para tarefas locais. Logo, o intervalo que deve ser usado para aplicações Multicast geralmente é o intervalo de 224.0.1.0 até 238.255.255.255.

Um grupo Multicast é formado pelo conjunto de *hosts* de uma rede que compartilham dados através de uma transmissão Multicast. Esses grupos são identificados por um endereço Multicast. Cada emissor envia pacotes para um endereço de grupo (por exemplo: 233.7.124.1), ao qual estão associados diversos receptores. Estes receptores, por sua vez, podem se vincular e desvincular de forma dinâmica ao grupo Multicast. Cabe aos dispositivos da rede e, em particular, aos roteadores determinar quais de suas interfaces possuem receptores participantes de um grupo Multicast e quais deverão receber uma cópia dos pacotes enviados para aquele grupo.

Dependendo da situação em que os *hosts* se encontram na rede, em relação ao suporte para transmissão Multicast, eles podem estar em três diferentes níveis de conformidade:

- Nível 0: é o nível sem suporte para o Multicast. Muitos *hosts* e roteadores na Internet estão nesta situação. Neste nível, os *hosts* não podem enviar e nem receber pacotes Multicast e ignoram estes pacotes.
- Nível 1: é o nível com suporte para envio de pacotes Multicast, porém sem suporte ao seu recebimento. Não é necessário ser membro de um grupo Multicast para poder enviar pacotes Multicast.
- Nível 2: é o nível com suporte completo para o Multicast. Neste nível os *hosts* podem enviar e receber pacotes via conexão Multicast. Basta saber como entrar e sair do grupo Multicast e como propagar a informação para os roteadores.

A seguir estão resumidos alguns benefícios do uso do Multicast:

- Desempenho Otimizado da Rede: o uso inteligente dos recursos da rede evita replicação desnecessária de pacotes. Uma melhor utilização do canal é obtida através de uma arquitetura mais eficiente para distribuição de dados, tornando melhor o desempenho da rede;

- Escalabilidade: serviços que usam o Multicast podem ser acessados por vários *hosts* ao mesmo tempo e podem aceitar novos membros no grupo a qualquer momento. A facilidade da inclusão de membros no grupo e a eficiência de transmissão tornam o Multicast um protocolo escalável.

2.1.1 Confiabilidade Multicast

A principal característica do Multicast é fato de o envio dos pacotes de um emissor para múltiplos receptores ser feita em uma única operação de transmissão. Portanto, não é viável o uso do protocolo TCP, uma vez que neste protocolo é necessário o estabelecimento de conexões ponto a ponto. Devido a esta característica, o Multicast é suportado pela camada de transporte através do protocolo UDP.

Cada pacote pode trafegar por um caminho diferente do emissor até o receptor, podendo chegar fora de ordem. Para resolver este problema, se faz necessário um algoritmo de ordenação de pacotes. Além do problema da entrega não ordenada dos pacotes, existe a possibilidade de perda de pacotes, tornando o protocolo Multicast não confiável.

Para resolver os problemas gerados pela não confiabilidade do protocolo Multicast, devido à transmissão UDP, é necessária a criação de mecanismos, no nível da aplicação, garantindo assim a confiabilidade na transmissão dos dados. A criação de uma camada acima da camada de redes, o protocolo Multicast Confiável, com a função de detectar perdas de pacotes e recuperar eficientemente os pacotes perdidos, torna a transmissão confiável.

As aplicações Multicast possuem diferentes exigências em relação à confiabilidade. Por exemplo, algumas aplicações exigem que a entrega dos dados seja feita com ordenação total, outras podem não fazer esta exigência. Assim como, algumas aplicações possuem muitos emissores de dados e outras, apenas um emissor. Em certas aplicações, todos os membros armazenam os dados enviados pelo emissor, podendo assim retransmitir um dado a qualquer momento, outras restringem a retransmissão ao emissor. Algumas aplicações exigem que o usuário tenha conhecimento de todas as informações trocadas durante uma sessão Multicast, outras permitem que o usuário entre na sessão a qualquer momento, sem a necessidade do conhecimento das informações trocadas antes de sua entrada [19].

Todas estas diferentes exigências que as aplicações fazem em relação a confiabilidade afetam o projeto do protocolo Multicast Confiável. A melhor maneira, segundo [19], de satisfazer todas as exigências das diversas aplicações é construir um protocolo Multicast que satisfaça o máximo de aplicações possíveis ao mesmo tempo, segundo uma definição mínima de confiabilidade Multicast e, deixar que as aplicações implementem suas funcionalidades específicas para garantir a confiabilidade na transmissão Multicast. A implementação de funcionalidades, como um algoritmo de ordenação, torna-se simples para a aplicação, por estarem sendo implementadas em cima do protocolo Multicast Confiável, que garante a confiabilidade de um modo geral.

Existem diferentes maneiras de se implementar um protocolo Multicast Confiável, já que a responsabilidade da detecção e da recuperação dos dados perdidos pode ser do emissor ou do receptor.

Nas Seções 2.2, 2.3 e 2.4 serão explicadas as três classes de protocolo Multicast Confiável:

Protocolo Sender-Initiated : onde as detecções de perdas são responsabilidade do emissor. Confirmações (*acknowledgments* ou ACKs) são enviadas pelos receptores e processadas pelos emissores.

Protocolo Receiver-Initiated : onde as detecções de perdas são responsabilidade do receptor. Os pedidos de retransmissões, NACKs (*negative acknowledgments*), são enviados via conexão Unicast para o emissor.

Protocolo Enhanced-Receiver-Initiated : onde as detecções de perdas também são responsabilidade do receptor, porém os NACKs são enviados via conexão Multicast para o grupo.

Na Seção 2.5 encontra-se a análise de desempenho destas classes, onde são avaliados a vazão e o atraso para recuperação dos dados, segundo estudos realizados em [36, 40]. Logo a seguir, na Seção 2.6, é feita uma análise do Multicast implementado na aplicação.

2.2 Sender-Initiated

O protocolo Sender-Initiated requer que o emissor mantenha uma lista dos ACKs recebidos para cada dado enviado, com a identificação dos receptores cuja confirmação (ACK) foi recebida pelo emissor. Todos os receptores, ao receberem um pacote de dados, enviam ao emissor um ACK, via conexão Unicast, informando que o pacote de dados foi recebido com sucesso. Este emissor atualiza a lista ACK, indicando que o receptor recebeu um determinado pacote enviado.

Segundo a transmissão de dados representada pela Figura 2.4, um servidor de dados assume o papel de emissor e transmite pacotes para os quatro membros receptores. As nuvens na figura representam o caminho que os pacotes percorrem entre um roteador e outro. Nota-se uma perda entre o roteador 1 e o roteador 2, ou seja, os membros C e D não recebem o pacote de dados enviados pelo servidor. Os membros A e B ao receberem o pacote, enviam os ACKs para o servidor (Figura 2.5).

A recuperação dos dados perdidos é tratada através do uso de temporizadores. O temporizador é ajustado para um intervalo de espera pelos ACKs de cada pacote no momento em que o emissor transmite um pacote de dados. Se todos os ACKs forem recebidos antes do intervalo, determinado pelo temporizador chegar ao fim, o emissor considera que todos os receptores receberam o pacote de dados com sucesso e o temporizador é desligado. Porém, se o intervalo expirar antes do emissor ter recebido os ACKs de todos os receptores, ele considera que um ou mais receptores não receberam o pacote de dados transmitido, retransmitindo, via conexão Multicast, o dado detectado como perdido e ajustando o temporizador para um novo intervalo de espera.

É necessário que o emissor tenha conhecimento de todos os receptores que estão

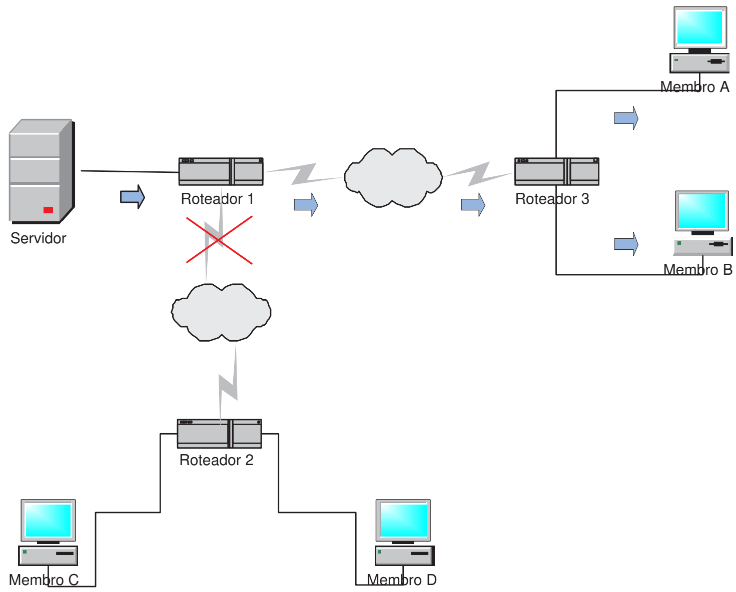

Figura 2.4: Transmissão e perda de dados

participando da sessão Multicast para controlar os ACKs recebidos. Portanto, as entradas e as saídas dos membros de uma sessão Multicast devem ser explicitamente notificadas ao emissor, evitando que haja retransmissões de dados desnecessariamente.

O emissor, ao receber um ACK, tem a certeza de que o receptor recebeu o pacote de dados enviado. Portanto, após o recebimento dos ACKs de todos os usuários, o emissor pode excluir este dado da memória. Esta é uma vantagem do protocolo Sender-Initiated. Porém, como a cada pacote enviado espera-se receber ACKs de todos os receptores, tem-se um aumento considerável do número de mensagens necessárias para manter a transmissão confiável, podendo sobrecarregar o emissor com o processamento dos ACKs recebidos. Quanto maior for o número de receptores, maior será o processamento no emissor, prejudicando a escalabilidade do protocolo Multicast, o que pode não ser interessante em algumas aplicações Multicast [36]. Além disso, como as mensagens de ACKs também estão sujeitas a perdas, podem ocorrer retransmissões desnecessárias dos pacotes. O protocolo XTP, apresentado em [35], é um exemplo de protocolo Sender-Initiated.

Devido a este problema de implosão de ACKs, dois outros protocolos foram

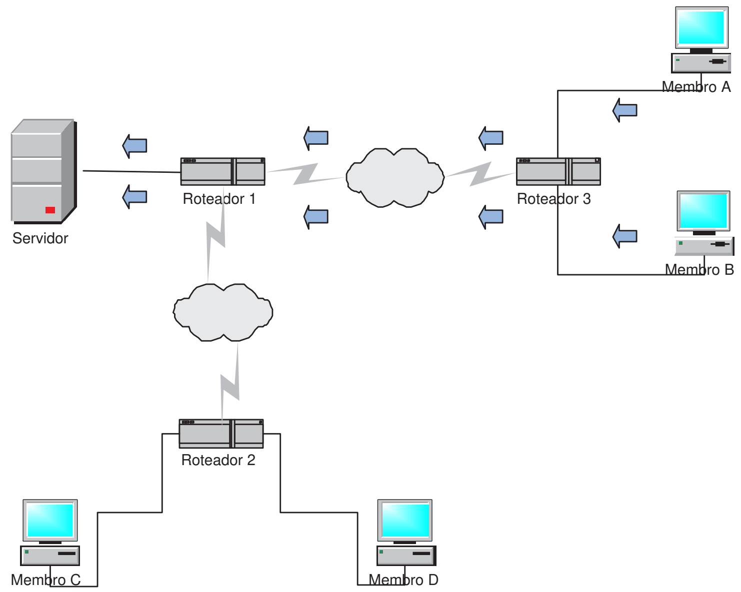

Figura 2.5: Sender-Initiated

propostos em [24, 27, 28], os protocolos baseados em árvore e em anel.

O protocolo baseado em árvore delega a responsabilidade da recuperação de dados a alguns receptores, organizando-os em uma árvore. Como pode ser observado na Figura 2.6, o emissor é a raiz da árvore e os receptores são divididos em grupos, ou seja, sub-árvores, onde as raízes são os receptores que assumem o papel de líderes. Os líderes se tornam responsáveis pelo recebimento dos ACKs locais do grupo de receptores da sub-árvore da qual ele é a raiz. Um receptor "filho" envia o ACK diretamente para o receptor líder, assim que o pacote de dados é recebido com sucesso. Note que este receptor "filho" também pode possuir receptores "filhos" , porém este não espera o recebimento dos ACKs de seus "filhos" para enviar o ACK ao receptor líder. O fato do receptor "filho" ter recebido o pacote de dados com sucesso, já garante aos seus "filhos" a retransmissão, caso algum deles tenha perdido o pacote. O ACK pode ser enviado ao receptor líder imediatamente. Porém, o dado deverá ser mantido na memória até que todos os receptores "filhos" tenham enviado seus ACKs, garantindo assim a retransmissão em caso de perda. Esta estrutura em árvore previne que receptores tenham contato direto com o emissor, mantendo a escalabilidade do protocolo. A escolha dos receptores líderes deve ser feita de maneira eficiente para que este mecanismo tenha um bom desempenho.

O protocolo *Tree-NAPP* (*NACK-avoidance and periodic polling*[33]) é uma subclasse do protocolo baseado em árvore, onde ACKs são enviados periodicamente confirmando os dados recebidos com sucesso e, NACKs são enviados requisitando dados perdidos. O líder envia as retransmissões apenas para a sub-árvore pela qual ele é responsável. O *Tree-based Multicast Transport Protocol* (TMTP[41]) é um exemplo de protocolo *Tree-NAPP*. Outro exemplo de protocolo baseado em árvore é o RMTP (*Reliable Multicast Transport Protocol*)([29])

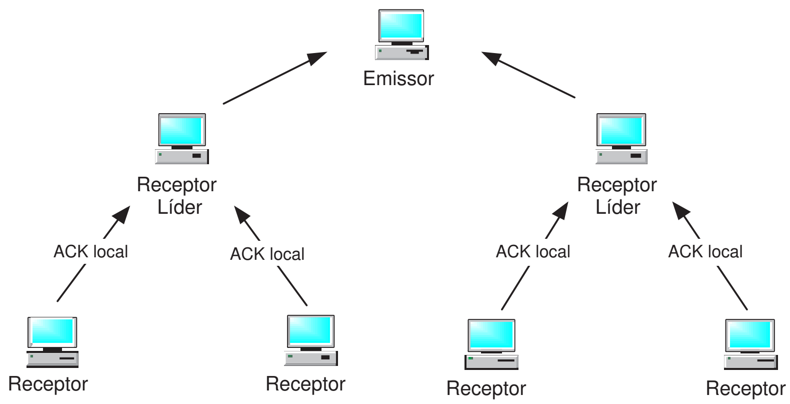

Figura 2.6: Um exemplo básico do protocolo baseado em árvore

O protocolo baseado em anel também tem como objetivo delegar a responsabilidade da recuperação de dados para os receptores. Este protocolo combina as vantagens do uso de NACKs com a confiabilidade do uso de ACKs. A cada momento somente um receptor estará com o *token*, tornando-o responsável pelo envio, via conexão Multicast, do ACK e pela retransmissão de dados, via conexão Unicast, caso seja necessário. Este *token* é passado através do anel de receptores. Se o tempo de espera por confirmação de pacote de dados recebidos com sucesso expirar antes do emissor ter recebido o ACK do receptor com o *token*, o emissor retransmite o dado perdido. O emissor somente apagará um dado da memória quando tiver recebido o ACK do receptor com o *token*, e neste momento ele passa a estar habilitado para envio de novos dados, ou seja, um novo dado somente é enviado após o recebimento do ACK. Caso os outros receptores detectem uma perda, devido ao recebimento de um pacote de dados fora de sequência, estes enviam um NACK, via conexão Unicast, para o receptor com o *token*. O *Reliable Multicast Protocol* (RMP[38]) é um exemplo de protocolo baseado em anel.

2.3 Receiver-Initiated

No protocolo Receiver-Initiated, a confiabilidade na transmissão dos dados é responsabilidade do receptor. Quando um receptor detecta que perdeu algum pacote de dados, ele envia ao emissor, via conexão Unicast, uma mensagem NACK, indicando que houve perda na transmissão dos dados e solicitando sua retransmissão. O emissor ao receber um NACK, retransmite o(s) dado(s) requisitado(s) via conexão Multicast.

Assumindo o mesmo cenário de perda representada pela Figura 2.4, os membros C e D detectam a perda ocorrida entre os roteadores 1 e 2 e enviam seus NACKs para o emissor (vide Figura 2.7).

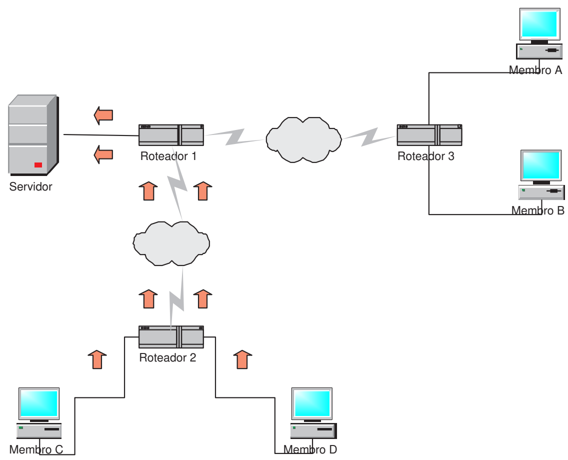

Figura 2.7: Receiver-Initiated

Para se precaver das perdas dos NACKs e das retransmissões, o protocolo Receiver-Initiated possui um processo de temporização análogo ao processo do protocolo Sender-Initiated. O temporizador é ajustado para um intervalo de espera pela retransmissão no momento em que o receptor envia o NACK. Se a retransmissão for recebida antes do intervalo de tempo chegar ao fim, o temporizador é desligado. Porém, se o intervalo expirar antes do receptor ter recebido a retransmissão do emissor, ele considera que houve uma perda da retransmissão ou da própria mensagem NACK enviada, retransmitindo o NACK e ajustando o temporizador para um novo intervalo de espera.

A perda de um pacote de dados é detectada a partir do número de sequência (*sn* - *sequence number*) que identifica os pacotes enviados pelo emissor. Quando um receptor recebe um pacote de dados, ele verifica se possui todos os pacotes com o *sn* inferior ao *sn* do pacote recebido. Por exemplo, se o receptor recebe um pacote com o sn = i +1 e verifica que não recebeu o pacote com o *sn* i, a perda é detectada. Esta forma de detectar a perda pode falhar caso o pacote de dados perdido seja o último enviado pelo emissor. Suponha que um membro A envie um pacote de dados com o $sn = 3$ e que um membro B tenha perdido este pacote. O membro B não detectará a perda até que receba um novo pacote de dados do membro A, mas se este não voltar a enviar novos dados, o membro B terá que encontrar uma outra forma de detectar a perda. Portanto, podem ser enviadas pelo emissor mensagens de controle notificando o *sn* do último pacote de dados enviado por ele.

Como a probabilidade de perda de pacotes, em geral, é menor que a probabilidade de sucesso no recebimento, a quantidade de mensagens de controle de recebimento (neste caso, os NACKs) enviadas pelos receptores ao emissor é menor do que no protocolo Sender-Initiated, diminuindo o *overhead* e reduzindo o processamento no emissor. O emissor só retransmite um dado caso tenha recebido um NACK, ou seja, não há retransmissões desnecessárias, o que tende a reduzir o número médio de retransmissões por pacote. Outra vantagem deste protocolo é o fato de que nenhum membro precisa ter o controle de todos os membros do grupo Multicast, podendo ter o conhecimento somente do membro emissor, pois será necessário para a retransmissão do pacote.

Apesar das vantagens descritas acima, também é possível analisar algumas desvantagens deste protocolo. Como o NACK é enviado pelo receptor, via conexão Unicast, somente o emissor é notificado da perda, restringindo a este, a responsabilidade da retransmissão do pacote perdido. Outra desvantagem é o fato do emissor não ter a certeza do recebimento dos dados por parte dos receptores, tornando-se complicado saber o momento em que um dado pode ser excluído da memória.

2.4 Enhanced-Receiver-Initiated

O protocolo Enhanced-Receiver-Initiated é uma variação do Receiver-Initiated. A confiabilidade na transmissão dos dados continua sob a responsabilidade do receptor, porém o NACK é enviado via conexão Multicast, acrescentando a este protocolo algumas vantagens significativas.

A perda de um pacote é detectada a partir do *sn* dos pacotes enviados pelo emissor, da mesma forma que é feito no protocolo Receiver-Initiated.

Os receptores enviam os NACKs via conexão Multicast, tornando possível a retransmissão do pacote por parte de qualquer um dos membros que tenha o pacote de dados requisitado. Distribuindo a responsabilidade de retransmissão dos dados, possibilita a redução do tempo de recuperação de um pacote caso o membro que retransmita o dado esteja mais próximo do requisitante do que o membro emissor. Por exemplo, se na transmissão representada pela Figura 2.4, a perda tivesse ocorrido entre o roteador 2 e o membro C, o membro D poderia retransmitir o dado antes do emissor.

O fato de vários membros poderem retransmitir simultaneamente o pacote solicitado, pode causar um excessivo aumento do tráfego. Assim como, quando vários receptores perdem o mesmo pacote de dados, vários NACKs são enviados pela rede, também podendo causar o aumento do tráfego, problema conhecido como implosão de NACKs.

Para evitar a implosão de NACKs, quando detectada a perda de um pacote, o receptor não envia o NACK imediatamente, mas somente após expirar o intervalo de tempo determinado pelo temporizador. Caso o receptor receba algum NACK, durante este intervalo de tempo, referente ao mesmo NACK que está escalonado, o envio do NACK é cancelado. Neste caso, o envio do NACK é suprimido, pois algum outro membro já fez a requisição de retransmissão para este pacote perdido, e o receptor passa a agir como se tivesse mandado o NACK, ou seja, fica esperando pela retransmissão durante um intervalo de tempo. Porém, se nenhum NACK for recebido até o intervalo de tempo expirar, então o receptor envia o NACK. Com este mecanismo de supressão de NACKs, os receptores distantes do emissor podem nunca precisar enviar um NACK, visto que os receptores mais próximos ao emissor podem detectar a perda e enviar NACKs antes dos receptores mais distantes, anulando o envio dos NACKs destes. Protocolos baseados no receptor que utilizam supressão de NAKs são denominados protocolos RINA (*Receiver Initiated with Nak Avoidance*)[27].

A Figura 2.8 mostra o processo de supressão de NACKs, supondo que os membros D e C disparam seus temporizadores para envio dos NACKs, e que o temporizador do membro C expira primeiro, enviando o NACK. O membro D ao detectar que o membro C já enviou um NACK para o mesmo pacote que pretendia enviar, cancela o envio do NACK.

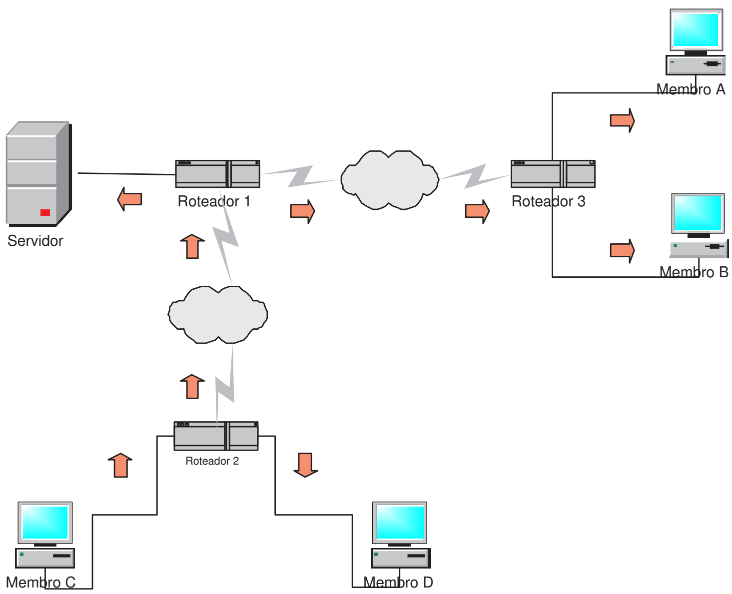

Figura 2.8: Enhanced-Receiver-Initiated

Após o envio do NACK, o receptor espera durante um intervalo de tempo pela retransmissão, como no protocolo Receiver-Initiated. Caso a retransmissão não chegue, o receptor considera que houve uma perda da retransmissão ou da própria mensagem NACK enviada e escalona o NACK para ser enviado novamente.

De maneira análoga ao processo de temporização feito para o envio de um NACK, uma retransmissão também não é enviada imediatamente, tentando-se evitar o aumento excessivo do tráfego. Quando um membro recebe um NACK requisitando um pacote de dados que ele tenha recebido, ele escalona o envio da retransmissão para algum tempo no futuro. Caso ele não receba nenhuma retransmissão para o mesmo pacote de dados até o intervalo de tempo expirar, a retransmissão é enviada. Do contrário, ele cancela o envio da retransmissão, pois algum outro membro já a fez.

Além das vantagens apresentadas pelo protocolo Receiver-Initiated, o protocolo Enhanced-Receiver-Initiated limita o número de NACKs enviados pela rede com o mecanismo de supressão de NACKs.

O SRM (*Scalable Reliable Multicast*[19]), protocolo genérico para transmissões Multicast confiável e o LBRM (*Log-based Receiver-Reliable Multicast*[22]) são exemplos de protocolos Enhanced-Receiver-Initiated.

Apesar das vantagens, a implementação destes métodos, onde a confiabilidade na transmissão é responsabilidade do receptor, é mais complexa do que no método onde o responsável pela confiabilidade é o emissor. Isso ocorre porque é necessário um bom ajuste dos temporizadores utilizados. Grandes valores para os temporizadores tendem a reduzir o tráfego gerado na rede, mas, por outro lado, aumentam o intervalo de tempo desde a emissão de um dado pelo emissor até a sua chegada, com sucesso, em todos os membros do grupo. O comportamento inverso pode ser observado se pequenos valores forem utilizados para os temporizadores. Um estudo preliminar deste comportamento foi feito em [14], onde foi construído um modelo com o objetivo de avaliar a sensibilidade das medidas de interesse (o número de retransmissões e o retardo para a recuperação dos pacotes) à variação dos intervalos dos temporizadores. No Capítulo 6 também será feita uma análise mais detalhada da sensibilidade das medidas de interesse à variação dos intervalos dos temporizadores.

2.5 Análise de Desempenho

Nas Seções anteriores foram apresentadas alternativas para a confiabilidade Multicast, baseadas no emissor e no receptor. Em [36, 40] são apresentados estudos analíticos onde são avaliados a vazão e o atraso para recuperação dos dados de cada uma destas alternativas.

Os modelos analíticos definidos para estes estudos possuem diversas simplificações, como por exemplo, assume-se a estrutura de um emissor e vários receptores, todos com a mesma probabilidade de perda. Segundo os experimentos observados em [39], supondo que a perda no backbone Multicast e a perda entre o emissor e o backbone são pequenas, é assumido que as perdas entre os receptores são independentes no espaço e no tempo. Supõe-se também que as transmissões dos ACKs e dos NACKs sejam confiáveis, nunca havendo perdas destas mensagens.

A análise da vazão em [36] avalia duas classes de aplicações: um-para-muitos, onde somente um membro envia os dados para os demais e muitos-para-muitos, onde todos os membros podem enviar e receber dados uns dos outros. O objetivo é computar a quantidade de tempo de processamento requerido para um pacote ser enviado com sucesso pelo emissor a todos os receptores. Este tempo de processamento inclui a quantidade de tempo necessária para enviar um dado, uma retransmissão, ACKs ou NACKs e os intervalos de tempo do temporizador antes do envio de alguns destes pacotes. A taxa máxima que um membro pode processar um pacote vai ser calculada de acordo com o tempo de processamento, e a vazão máxima de um protocolo pode ser determinada a partir destas taxas de processamento.

O desempenho do protocolo Sender-Initiated é dependente do número de membros no grupo Multicast, pois com o aumento do número de membros, a taxa de processamento dos emissores aumenta, devido ao aumento dos ACKs que terão que ser processados. Portanto, os protocolos baseados no receptor são mais escaláveis uma vez que a taxa de processamento é independente do número de membros.

Em [36] foi demonstrada a superioridade de desempenho dos protocolos Receiver-Initiated e Enhanced-Receiver-Initiated em relação ao protocolo Sender-Initiated.

O protocolo Enhanced-Receiver-Initiated tem o melhor desempenho para aplicações um-para-muitos, enquanto o protocolo Receiver-Initiated tem o melhor desempenho para aplicações muitos-para-muitos. Isso se deve ao fato do envio dos NACKs, via conexão Multicast, adicionar uma complexidade nas funcionalidades do receptor, que no caso das aplicações muitos-para-muitos ainda tem a funcionalidade de enviar dados.

A pesquisa feita em [7] confirma os estudos citados anteriormente sobre o desempenho dos protocolos segundo análise da vazão, acrescentado o estudo do desempenho dos protocolos baseados em árvore. Nos resultados obtidos foi mostrado que os protocolos baseados em árvore têm melhor desempenho que os demais, ficando o protocolo Enhanced-Receiver-Initiated em segundo lugar.

O foco de análise em [40] é o comportamento do atraso em relação ao número de receptores, a propagação do atraso na rede e aos valores dos temporizadores. Visto que o atraso pode ser influenciado por muitos fatores, incluindo o tempo de processamento de um pacote pelos membros, o tempo de propagação na rede, a probabilidade de perda dos pacotes e os valores escolhidos para os temporizadores. Sob esta análise, mais uma vez foi mostrada a superioridade do Enhanced-Receiver-Initiated em relação aos protocolos Receiver-Initiated e Sender-Initiated.

2.6 Multicast na Aplicação

Existem muitos projetos de pesquisa com o objetivo de construir uma aplicação que provê serviços Multicast sem requerer o suporte direto dos roteadores, transferindo a responsabilidade do Multicast para a aplicação [10, 12, 13, 20, 32, 42].

Algumas características do IP Multicast levaram à implementação de aplicações Multicast sobre o protocolo IP Unicast. São elas: o roteador precisa manter uma tabela com o estado de todos os membros do grupo Multicast, introduzindo complexidade e problemas de escalabilidade ao roteador; qualquer emissor pode enviar uma mensagem para um grupo Multicast, tornando a rede vulnerável a ataques de emissores maliciosos, complicando o seu gerenciamento; prover confiabilidade, controle de congestionamento, controle de fluxo e segurança tem sido mais difícil do que no caso do Unicast, que usa o protocolo TCP.

Estas aplicações, provedoras de serviços Multicast, podem ser chamadas de *End System* Multicast [12, 13]. Os membros do grupo Multicast se organizam em uma árvore para entrega dos dados. Cada aresta da árvore corresponde a uma conexão Unicast TCP entre dois membros do grupo. O protocolo TCP provê uma conexão confiável [23], com segmentação dos dados a serem enviados pela aplicação, controle de recebimento através de mensagens ACKs enviadas pelo receptor e controle de fluxo através do uso de *buffers*, onde o receptor avisa ao emissor o quanto de espaço nos *buffers* ele tem disponível para recebimento dos dados.

A Figura 2.9 mostra um exemplo de *End System* Multicast, onde os membros A, B, C e D são os sistemas finais (*end systems*).

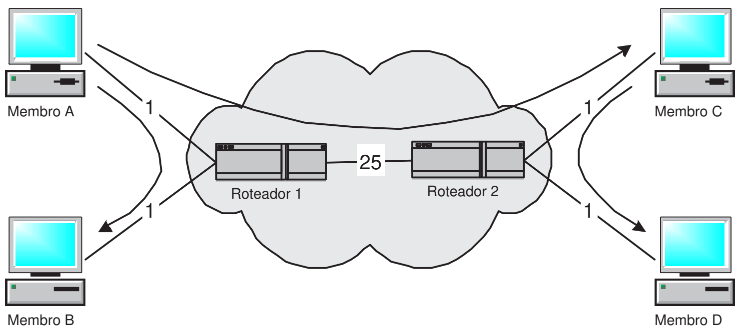

Figura 2.9: Exemplo de *End System* Multicast

Um dos desafios das aplicações Multicast é prover métricas para que os membros se organizem na árvore de forma eficiente, adaptando-se às mudanças da rede e mantendo o desempenho da aplicação. Cada protocolo pode possuir métricas variadas para a organização de forma dinâmica dos membros na árvore.

Os protocolos propostos em [13, 20] apenas consideram o atraso como métrica de adaptação, enquanto os protocolos propostos em [12, 42] além de considerarem o atraso também consideram o *stress* no canal como métrica, ou seja, o número de pacotes duplicados trafegando pelo canal.

Nas aplicações *End System* Multicast sempre haverá pacotes duplicados no canal, visto que é impossível evitar que as arestas da árvore passem pelo mesmo canal físico da rede. Por exemplo, a Figura 2.9 mostra que o uso de Multicast na aplicação diminui o número de pacotes por canal, em relação à transmissão Unicast nativa, que exigiria o envio de três pacotes de dados no canal entre o emissor A e o roteador 1, e que com o uso do Multicast na aplicação este número diminui para dois, porém ainda mantém a duplicação no canal.

Ainda na Figura 2.9, podemos perceber o aumento do atraso entre os membros do grupo em relação ao IP Multicast. Se a transmissão tivesse sido feita via IP Multicast, ou seja, diretamente do emissor A para o receptor D, o atraso seria de 27 unidades de tempo. Porém, com o uso do Multicast na aplicação o atraso aumenta para 29, uma vez que o emissor A envia o pacote de dados para o receptor C e este repassa o pacote para o receptor D.

Os principais pontos a serem considerados, na solução proposta, são o aumento do atraso e o número de pacotes duplicados atravessando os canais da rede. Com isso, a implementação do Multicast na aplicação pode não ser aconselhável em casos onde grandes atrasos não são toleráveis e em situações onde a duplicação de pacotes nos canais é prejudicial ao desempenho da rede.

Capítulo 3

A Aplicação Tangram II Whiteboard

NESTE capítulo é apresentado o Tangram II Whiteboard, uma ferramenta distribuída, que requer o uso de uma transmissão Multicast confiável. São descritas algumas características desta ferramenta com o objetivo de ressaltar suas vantagens, as quais não são encontradas nas demais ferramentas disponíveis atualmente.

3.1 Introdução

O crescimento e o progresso das redes de computadores facilitaram o desenvolvimento de ferramentas com aplicação distribuída. O objetivo destas ferramentas é manter a comunicação entre os usuários em diferentes localizações na rede. O Tangram II Whiteboard (TGWB) é um exemplo deste tipo de ferramenta, cuja finalidade é permitir que um grupo de usuários, em locais distintos, compartilhe a mesma área gráfica. As aplicações do TGWB são muito amplas, podendo ser usado em teleconferências e no ensino à distância, ou até mesmo no desenvolvimento em equipe de sistemas à distância [14]. A Figura 3.1 ilustra as interfaces de dois usuários da ferramenta TGWB, onde tudo que foi desenhado na primeira interface foi recebido com sucesso pela segunda interface.

Um dos pontos cruciais para o desenvolvimento de uma ferramenta *whiteboard* é

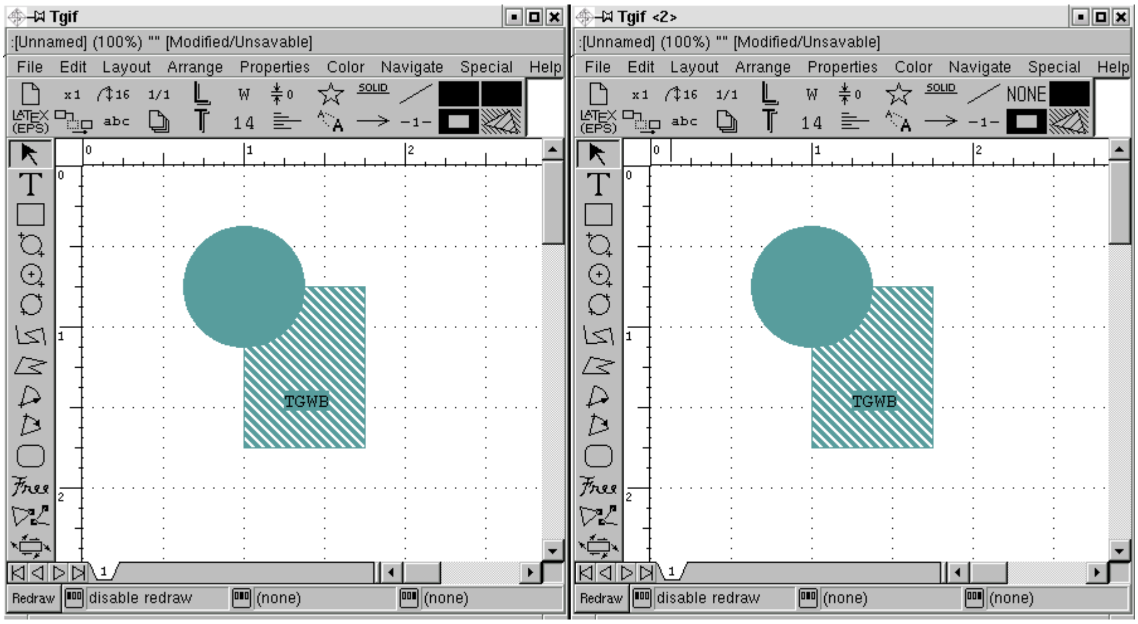

Figura 3.1: Interface TGWB

a garantia da consistência entre as informações de cada um dos usuários. Portanto, duas funcionalidades devem estar presentes na implementação de uma ferramenta *whiteboard*:

1. a detecção e a recuperação eficiente dos dados perdidos;
2. a execução dos comandos gráficos na mesma ordem para todos os usuários do grupo.

Para garantir a consistência dos dados, segundo a primeira funcionalidade descrita acima, a ferramenta *whiteboard* precisa utilizar um mecanismo de comunicação Multicast confiável. Sendo assim, o TGWB utiliza a biblioteca Multicast Confiável (RML), desenvolvida no âmbito desta pesquisa, para garantir que a comunicação de dados seja feita de forma confiável, ou seja, que todos os usuários recebam as mesmas informações. A biblioteca RML é apresentada com detalhes no Capítulo 4.

A segunda funcionalidade envolve o conceito de ordem entre eventos distribuídos. Se cada usuário executar o mesmo conjunto de comandos recebidos em ordem diferente, eles chegarão a um estado de inconsistência. A Figura 3.2 mostra um exemplo de inconsistência de dados gerada devido à execução de comandos em ordens diferentes. Neste exemplo, o usuário A recebe os desenhos na seguinte ordem: primeiro o triângulo, depois o quadrado e por último o texto "X". E o usuário B recebe os desenho em uma ordem diferente do usuário A: primeiro o triângulo, depois o texto "X" e por último o quadrado. A ordem de recebimento e execução dos desenhos pelo usuário B faz com que o texto fique atrás do quadrado, criando um desenho diferente do usuário A.

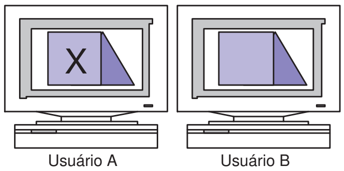

Figura 3.2: Exemplo de inconsistência de dados entre dois usuários

Para evitar o problema da inconsistência, o TGWB implementou um algoritmo de ordenação única dos comandos recebidos para todos os membros do grupo.

Existem diversos trabalhos sobre *whiteboard* na literatura. A maioria destes trabalhos disponibilizam recursos limitados para edição gráfica, além de restringirem o comportamento do usuário para simplificar problemas relacionados à ordem de execução dos comandos. Em [19] é apresentado um *whiteboard*, onde o problema de consistência entre os estados dos usuários é simplificado, não permitindo que um usuário modifique os objetos criados por outros usuários. Desta forma, o aplicativo não aborda o problema de ordenação dos comandos.

O *whiteboard* apresentado em [37] também não permite que um usuário modifique os objetos criados por outro. Neste *whiteboard* ([37]), conhecido por *MediaBoard*, quando um novo usuário deseja entrar na sessão Multicast, ele não será capaz de recuperar os dados de usuários que já saíram da sessão. Suponha uma sessão Multicast inicialmente com os usuários A e B, e que o usuário C entre no grupo depois que os primeiros usuários tenham trocado algumas informações e que o usuário A já tenha saído da sessão. O usuário C não será capaz de recuperar os dados enviados pelo usuário A, pois o usuário B retransmitirá somente seus próprios dados, tornando as interfaces dos usuários B e C inconsistentes. Uma outra característica do *MediaBoard* é a apresentação imediata na interface dos dados recebidos, mesmo havendo perdas no recebimento. Por exemplo: se ao receber o texto "Recebimento coerente dos dados" a palavra "coerente" é perdida, mesmo assim o restante da frase é mostrado na interface. A perda de letras também não impede que o texto apareça na interface antes da recuperação total da frase. Por exemplo: se a letra "a" da palavra "dados" for perdida, a palavra "ddos" aparecerá na interface até que a letra "a" perdida seja recuperada e a palavra ou a frase correta seja mostrada na interface. O TGWB somente mostra um objeto na interface quando todos os dados perdidos, referentes a este objeto, forem recuperados, de modo que o objeto apareça por inteiro na interface, e não apenas parte dele.

Durante os testes feitos com o objetivo de testar seu desempenho, o *MediaBoard* se mostrou bastante instável. Por exemplo quando um usuário sai do grupo Multicast e depois decide entrar novamente, a execução da ferramenta de todos os demais usuários do grupo é cancelada imediatamente.

Outro exemplo de ferramenta *whiteboard* é o *Microsoft NetMeeting*, onde o Multicast é simulado, ou seja, múltiplas conexões Unicast são estabelecidas para o envio dos dados a um grupo de receptores [3, 14]. Como foi visto no Capítulo 2, o uso de Multicast simulado pela aplicação pode não ser a maneira mais eficiente de distribuição de dados a um grupo de receptores, uma vez que pode aumentar o número de pacotes no canal e o atraso na entrega dos pacotes.

3.2 O Whiteboard TGWB

A ferramenta TGWB mantém interface com o usuário através do editor gráfico TGIF (*Tangram Graphic Interface Facility* [11]). O TGIF é um poderoso aplicativo para edição gráfica baseada em objetos. Neste aplicativo um desenho é criado a partir de objetos primitivos, tais como: linhas, circunferências, polígonos, etc. Um objeto pode ser construído a partir do agrupamento de outros objetos. Os objetos podem possuir vários tipos de textura e suportar diversas alterações como rotação e escala, além de suportar a criação e a edição de texto, permitindo a criação de slides e transparências para apresentações. O TGIF também é usado pela ferramenta de modelagem e análise de sistemas, TANGRAM II, que será apresentada no Capítulo 4. O TGWB faz parte do conjunto de ferramentas TANGRAM II que, juntamente com um aplicativo de voz desenvolvido [17, 18] e um servidor de vídeo [8], fornecerão um conjunto poderoso de ferramentas de teleconferência e de ensino à distância.

A ferramenta TGWB permite que os usuários modifiquem o mesmo objeto, ou seja, se o usuário A desenhar um quadrado, não somente ele, mas qualquer outro usuário do grupo pode modificar este quadrado, seja aumentando seu tamanho, mudando sua cor, ou até mesmo removendo o quadrado da interface. Uma ferramenta que possui esta característica deve possuir um algoritmo de ordenação dos dados para que os comandos recebidos dos diversos usuários sejam executados na mesma ordem para todos eles, mantendo a consistência global do dados. Assim, o quadrado citado anteriormente deverá chegar ao mesmo estado final em todas as máquinas, depois de todas as modificações sofridas pelos diversos usuários.

O funcionamento do TGWB é baseado em trocas de mensagens. Todo comando executado por um usuário na interface gráfica local (TGIF) é codificado e enviado a todos os demais usuários do grupo através de uma mensagem. Quando uma mensagem é recebida, o comando é decodificado e executado localmente como se fosse um comando gerado pelo próprio usuário receptor da mensagem. Desta forma, todos os comandos gerados pelos usuários em uma sessão são executados localmente no aplicativo TGWB de cada um. As mensagens do TGWB são trocadas através da biblioteca Multicast RML (a ser apresentada no Capítulo 4).

O TGWB também faz a segmentação dos pacotes antes do envio dos dados trocados em uma sessão Multicast, devido à limitação em relação ao tamanho dos pacotes que podem trafegar pela rede. Nem todos os roteadores da rede possuem um algoritmo de segmentação de pacotes, eliminando os pacotes que excedem o limite de tamanho suportável pelo roteador. Portanto, se faz necessário um algoritmo de segmentação dos dados antes de seu envio pela rede. Esta segmentação garantirá que os pacotes possam passar em todos os roteadores da rede sem o risco de serem eliminados por causa do seu tamanho. Se um usuário deseja enviar uma imagem muito grande para o grupo, esta imagem deve ser segmentada em pacotes suficientemente pequenos antes de ser enviada pela rede. O tamanho do segmento é escolhido de modo que a maioria dos objetos caibam em apenas um segmento e que, ao mesmo tempo, este segmento seja menor que a unidade máxima de transferência (MTU).

As mensagens trocadas durante uma sessão são recebidas através do IP Multicast, e a confiabilidade na transmissão Multicast é tratada pela biblioteca RML. No aplicativo TGWB é feita a segmentação/remontagem e ordenação dos dados recebidos para serem mostrados na tela do usuário, via interface TGIF. A Figura 3.3 demonstra as camadas do TGWB.

Figura 3.3: Camadas do TGWB Whiteboard

3.3 Ordenação dos Comandos no TGWB

Nesta seção é descrito o processo de ordenação dos comandos. Uma vez que a camada de confiabilidade Multicast é implementada abaixo da camada de ordenação de comandos (vide Figura 3.3), o TGWB assume que os dados recebidos são confiáveis, ou seja, todas as perdas já foram recuperadas anteriormente.

A consistência de dados entre os usuários é garantida se todos os usuários executarem os comandos recebidos na mesma ordem. A abordagem utilizada para resolução do problema de inconsistência de dados é a mesma usada para análise de uma computação distribuída genérica [5]. Os usuários representam os nós desta computação e os comandos são eventos gerados por estes nós. Desta forma, o problema se resume à ordenação de eventos gerados distribuidamente.

Formalização do Problema de Inconsistência de Dados

A computação distribuída pode ser analisada como a execução de um conjunto de eventos, denotado por $\mathcal{E}$, onde os eventos são considerados a unidade fundamental da computação. No caso do TGWB, o evento é representado pela geração de um comando na interface da ferramenta e pode ser definido pela tupla ϵ [14]:

$$\epsilon = (n_i, \tau, \Lambda, m)$$

onde

- n_i é o nó (usuário) em que o evento (comando) foi gerado.
- τ é o tempo em que ocorre o evento, dado pelo relógio local do nó n_i.
- Λ é o conjunto de mensagens (geradas por outros nós) recebidas por n_i, que influenciam diretamente a geração do evento ϵ.
- m é a mensagem enviada para outros nós em decorrência do evento ϵ.

A definição de evento apresentada através da tupla ϵ é bastante geral, onde a definição do conjunto de mensagens Λ fica a critério da aplicação distribuída que está empregando a definição. No TGWB este conjunto de mensagens Λ pode ser definido por comandos que se referem ao mesmo objeto ou que atuam sobre a mesma área gráfica na interface.

Uma vez que a mensagem m, enviada por um nó ao gerar um evento, pode pertencer ao conjunto de mensagens Λ que influenciam a geração de eventos em outros nós, dizemos que os eventos em $\mathcal{E}$ estão fortemente inter-relacionados. A relação, denotada por $\longrightarrow$, é definida sobre o conjunto de eventos $\mathcal{E}$:

Definição 1 : *Se ϵ_1 e ϵ_2 são eventos, então $\epsilon_1 \longrightarrow \epsilon_2$ se, e somente se uma das seguintes condições é satisfeita*

i) ϵ_1 e ϵ_2 são gerados pelo mesmo nó, respectivamente, com tempos locais τ_1 e τ_2, tal que $\tau_1 < \tau_2$. E nenhum outro evento ocorre neste nó no tempo τ, tal que $\tau_1 < \tau < \tau_2$

ii) ϵ_1 e ϵ_2 são gerados em nós distintos e a mensagem m, enviada em decorrência de ϵ_1, pertence ao conjunto de mensagens Λ de ϵ_2

A condição *i)* expressa o conceito intuitivo de causalidade que existe entre eventos gerados pelo mesmo nó, enquanto que a condição *ii)* representa a relação de causa-efeito existente entre eventos gerados por nós distintos. A Figura 3.4 exibe o grafo de precedência $H = (\mathcal{E}, \longrightarrow)$, definido pela relação $\longrightarrow$, para a execução de uma computação distribuída. O grafo H é um grafo acíclico direcionado, em que o conjunto de nós é definido pelo conjunto de eventos $\mathcal{E}$, e suas arestas são determinadas pelos pares de eventos relacionados por $\longrightarrow$.

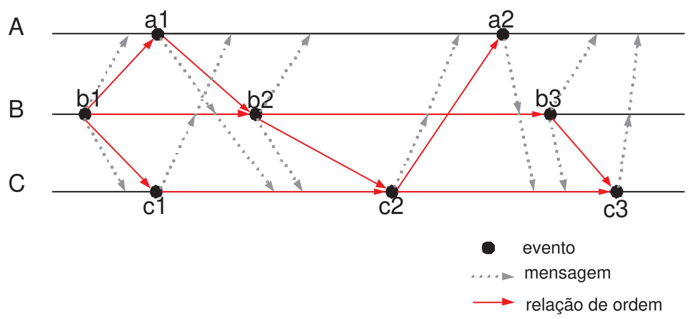

Figura 3.4: Grafo de precedência para uma computação distribuída

A denotação $\longrightarrow^+$ estabelece uma ordem parcial para o conjunto de eventos $\mathcal{E}$. Dois eventos estão relacionados por $\longrightarrow^+$ se existir um caminho entre eles no grafo de precedência H. O par de eventos $(a1, c3)$ da Figura 3.4 é um exemplo de eventos relacionados por $\longrightarrow^+$, pois o evento $c3$ só pode ser executado depois que acontecer o evento $a1$. Dois eventos ϵ_1 e ϵ_2 que não estejam relacionados por $\longrightarrow^+$ são ditos eventos concorrentes. Os eventos $a1$ e $c1$ da Figura 3.4 são exemplos de eventos concorrentes.

A relação $\longrightarrow^+$ pode ser usada para definir os conceitos de passado e futuro de um evento.

Definição 2 : *Para um evento ϵ, definimos*

$$Passado(\epsilon) = \{\epsilon' | \epsilon' \longrightarrow^+ \epsilon, \epsilon' \in \mathcal{E}\}$$

$$Futuro(\epsilon) = \{\epsilon' | \epsilon \longrightarrow^+ \epsilon', \epsilon' \in \mathcal{E}\}$$

Inicialmente bastaria executar os comandos de acordo com os conjuntos *Passado(ϵ)* e *Futuro(ϵ)*, definidos acima, para garantir a consistência entre os usuários de uma sessão TGWB, utilizando-se a seguinte regra:

Regra 1 : *"O comando associado ao evento ϵ deve ser executado após a execução de todos os eventos pertencentes ao conjunto Passado(ϵ), e antes da execução dos eventos pertencentes ao conjunto Futuro(ϵ)."*

Porém, os conjuntos *Passado(ϵ)* e *Futuro(ϵ)* foram definidos a partir da relação de ordem parcial $\longrightarrow^+$. Ainda podem existir eventos que não pertençam a nenhum destes conjuntos, ou seja, os eventos concorrentes ao evento ϵ. Se for estabelecida uma ordem única, a ser utilizada por todos os nós da sessão, para estes eventos concorrentes, então, pode ser alcançado um estado de consistência global entre todos os usuários.

Suponha o exemplo de inconsistência representado pela Figura 3.2 e detalhado pela Figura 3.5, que representa uma sessão Multicast com três usuários. O usuário A desenha um triângulo na sua interface e, consequentemente, é enviada aos demais usuários da sessão uma mensagem contendo este comando. Após o recebimento do comando de desenho do triângulo ter sido recebida pelos usuários B e C, estes fazem, simultaneamente, desenhos distintos em suas interfaces. O desenho do quadrado e o texto "X", criados por B e C, respectivamente, geram eventos concorrentes. Caso o critério utilizado para definir a ordem de execução de eventos concorrentes seja a ordem de chegada das mensagens, os usuários chegariam a um dos estados de inconsistência representados pela Figura 3.5, onde o estado final do usuário A depende de qual comando chegou primeiro, o comando vindo do usuário B ou o comando vindo do usuário C.

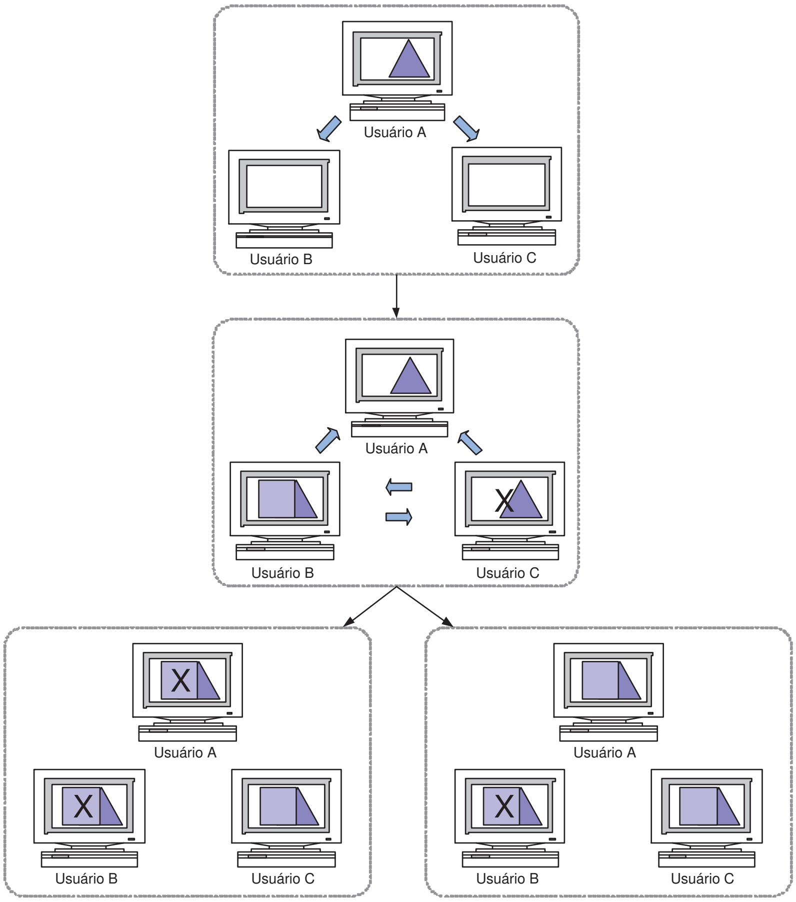

Figura 3.5: Exemplo de inconsistência de dados entre três usuários

Para resolver este problema , é necessário ampliar $\longrightarrow^{+}$ para obter uma relação de ordem total representada por $\longrightarrow_{t}^{+}$. Uma ordem total é uma ordem parcial em que, para todo ϵ_1, $\epsilon_2 \in \mathcal{E}$, existe uma das relações (ϵ_1, ϵ_2) ou (ϵ_2, ϵ_1). A ordem total é obtida a partir de $\longrightarrow^{+}$, incluindo os pares de eventos concorrentes. Os eventos concorrentes não existem quando a ordem total é aplicada. Portanto, os conjuntos *Passado*(ϵ) e *Futuro*(ϵ) são redefinidos de acordo com esta nova relação e a Regra 1 passa a garantir a consistência entre os usuários do TGWB.

Relógios Lógicos

A implementação da ordenação total de eventos é feita através de relógios lógicos, utilizando o algoritmo apresentado por Lamport em [25]. Um relógio lógico é apenas uma maneira de associar um número a um evento. A definição de relógio lógico deve ser baseada na relação de ordem entre os eventos, de acordo com a seguinte condição:

Condição 1 : *Para quaisquer dois eventos a e b, se* $a \longrightarrow^{+} b$*, então* $C(a) < C(b)$.

onde C(ϵ) é o valor do relógio associado ao evento ϵ.

É necessário que cada nó n_i mantenha um contador C_i, de modo que no momento da geração de um evento ϵ, $C_i(\epsilon)$ seja o valor do contador C_i neste instante. Os contadores C_i são atualizados de acordo com as seguintes regras:

R1: Imediatamente após a geração de um evento, o nó n_i incrementa seu contador de uma unidade.

R2: Ao receber uma mensagem, o nó n_i ajusta o seu contador para o valor

$$\max\{C_i, C(msg) + 1\} \tag{3.1}$$

onde $C(msg)$ é o valor do contador do emissor da mensagem no momento de sua geração.

Este mecanismo oferece apenas uma relação de ordem parcial entre os eventos. Como visto anteriormente, esta relação é insuficiente para garantir a consistência entre os usuários. Portanto, é necessário estender este mecanismo para que ele suporte a relação de ordem total. Se os valores dos relógios lógicos forem iguais, então, os eventos são concorrentes, por exemplo, os eventos a e b são concorrentes se e somente se $C(a) = C(b)$. Isto só é possível se estes eventos forem gerados por nós distintos. Para não haver concorrência entre os eventos, é necessário que cada nó n_i possua uma identificação única, $Id(n_i)$, que pode ser por exemplo a combinação do endereço IP da máquina com o número do processo (PID). Neste caso, pode ser definido uma ordem total para estes nós, segundo a Definição 3.

Definição 3 : *Sejam a e b eventos gerados em n_i e n_j respectivamente. Então, $a \longrightarrow_t^+ b$ se, e somente se, uma das seguintes condições é satisfeita*

i) $C(a) < C(b)$

ii) $C(a) = C(b)$ e $Id(n_i) < Id(n_j)$

Algoritmo para Execução dos Comandos

Um algoritmo simples de execução dos comandos pode ser definido segundo a Regra 1 apresentada nesta seção, onde todo comando gerado ou recebido seja armazenado até que todos os comandos pertencentes ao seu passado tenham sido recebidos e executados, quando então ele poderá ser executado. Porém, a execução deste simples algoritmo apresenta dois problemas:

1. a lentidão para a aplicação;

2. a necessidade de detectar se todo o passado do comando já foi executado.

O primeiro problema consiste no retardo para execução de um comando, que não pode ser executado imediatamente após a sua geração, ou no recebimento, uma vez que deve esperar até que todos os comandos em seu passado sejam executados. Esta espera pode aumentar significativamente o tempo de latência entre a geração do comando por um usuário e a sua execução em todas as interfaces. Além disso, este algoritmo apresenta uma situação indesejável ao usuário, pois os comandos que ele gera em sua própria interface não são imediatamente executados, mas sim armazenados para futura execução. Certamente, esta não é uma situação desejável para um aplicativo interativo como o *whiteboard*.

O segundo problema consiste em detectar se todo o passado de um determinado comando c já foi executado. Este problema é um pouco mais complexo, pois os comandos pertencentes ao passado de c, gerados localmente, já se encontram todos executados ou armazenados. Porém, os comandos pertencentes ao passado de c, gerados por outros usuários, tornam-se difíceis de serem detectados.

Devido aos dois problemas apresentados acima, foi implementada uma solução alternativa para a execução dos comandos [14]. O algoritmo implementado leva em consideração duas observações importantes: uma é que em muitos casos, apesar da relação $\longrightarrow_t^+$ determinar que um comando c_1 está no passado de um outro comando c_2, a troca da execução entre eles não afeta o resultado final. Por exemplo, se os comandos se referirem a objetos distintos e atuarem sobre áreas diferentes na interface gráfica. Outra observação é que a interface do TGWB, o editor gráfico TGIF, fornece o recurso para desfazer a execução dos comandos. Desta forma, os comandos podem sempre ser desfeitos e executados novamente em uma ordem diferente. Esta operação é chamada *RollBack and Recovery*.

O algoritmo de execução dos comandos foi definido levando em consideração as duas observações descritas acima. A idéia principal é tentar executar os comandos imediatamente após seu recebimento. Porém, quando o comando recebido se refere a algum objeto inexistente, este comando não pode ser executado imediatamente, sendo armazenado para futura execução e caracterizado como inconsistente.

Se ao chegar um novo comando for detectado que a ordem de execução utilizada transgride a ordem $\longrightarrow_t^+$, ou seja, o valor do C_i da mensagem recebida for menor que o valor do C_i do último evento executado, então é feita a operação *RollBack* até o ponto onde o novo comando deve ser executado. O novo comando é executado e os comandos que foram desfeitos são novamente executados através da operação *Recovery*. Para isto, cada usuário deve manter uma lista com todos os comandos conhecidos por ele. Esta lista é ordenada de acordo com o sistema de relógios lógicos e cada comando da lista pode se encontrar no estado executado ou inconsistente, como representado na parte A da Figura 3.6.

Quando um novo comando é recebido pelo usuário, seu ponto de inserção na lista de comandos é identificado. Todos os comandos que se encontram antes deste ponto pertencem ao passado do novo comando e todos que se encontram depois deste ponto pertencem ao seu futuro (vide parte B da Figura 3.6). De acordo com a Regra 1, o novo comando deve ser executado antes de todos os comandos em seu futuro. Portanto, todos os comandos da lista que pertencem ao seu futuro são desfeitos e o novo comando executado e os comandos que foram desfeitos são novamente executados ou, finalmente executados, caso estivessem anteriormente em estado inconsistente (vide parte C da Figura 3.6).

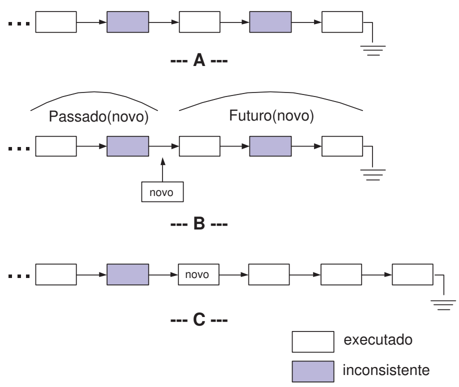

Figura 3.6: Comandos

Cada usuário do grupo envia periodicamente uma mensagem, informando qual o último comando gerado por ele. Desta forma, os demais usuários podem detectar se já receberam todas as mensagens que pertencem ao passado de um determinado comando, podendo então eliminá-los da lista. Com isso, pode ser evitado o crescimento desnecessário da lista de comandos.

Capítulo 4

A Biblioteca Multicast Confiável

NESTE capítulo será apresentado o funcionamento da biblioteca Multicast Confiável, desenvolvida com o objetivo de recuperar eficientemente descartes de pacotes na rede durante a transmissão. Inicialmente será explicado o funcionamento do protocolo, referente à entrada e à saída de membros em uma sessão Multicast. Em seguida, é apresentada uma das estruturas de armazenamento dos dados (a *cache*). E por fim, algumas características da biblioteca são descritas, dentre elas: o mecanismo de detecção e recuperação de dados perdidos, a lista de eventos e o uso dos temporizadores para a supressão dos NACKs e das retransmissões.

4.1 Introdução

Conforme visto no Capítulo 2, é necessária a implementação da confiabilidade na transmissão Multicast. Após a realização de pesquisas na Internet com o objetivo de encontrar alguma biblioteca que tratasse da transmissão Multicast confiável, não foi encontrada nenhuma biblioteca disponível que funcionasse sem problemas. Com isso, foi desenvolvida uma biblioteca com funções para garantir a confiabilidade na transmissão Multicast. Esta biblioteca, chamada de *Reliable Multicast Library* (RML), pode ser vista como um protocolo em uma camada superior à camada de transporte, com a função de detectar perdas de pacotes e recuperar eficientemente os pacotes perdidos, tornando a transmissão confiável. Qualquer aplicação que precisa da confiabilidade na transmissão Multicast pode usar a biblioteca RML.

A biblioteca foi desenvolvida no laboratório de modelagem, análise e desenvolvimento de redes e computação de sistemas (LAND), da Universidade Federal do Rio de Janeiro (UFRJ), obra de um trabalho de grupo que envolveu a atuação de mais dois alunos, um de mestrado e outro de graduação. Sua distribuição é gratuita pela Internet [1].

A Figura 4.1 ilustra como a biblioteca RML faz a interface entre a aplicação e a rede.

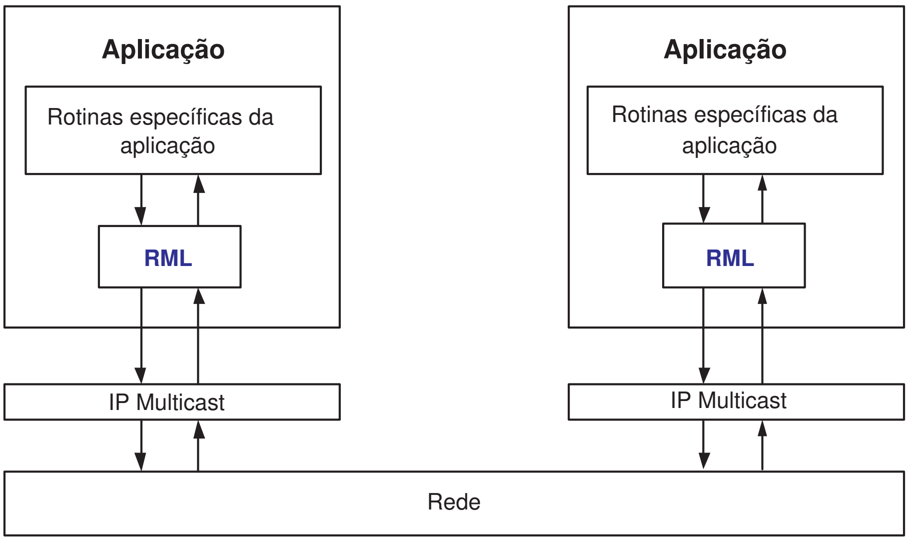

Figura 4.1: A camada RML

A biblioteca RML, implementada na linguagem de programação C, é baseada no protocolo Enhanced-Receiver-Initiated [27, 36, 40], devido à superioridade de seu desempenho em relação aos demais protocolos. Com isso fica atribuída ao receptor a responsabilidade da recuperação dos dados perdidos. Esta biblioteca mantém a comunicação entre os membros do grupo Multicast de maneira transparente, onde os membros do grupo não precisam conhecer a topologia da rede. Além disso, a biblioteca possui facilidades para que os membros possam entrar e sair do grupo de forma dinâmica.

4.2 A *Cache*

A estrutura *cache* foi implementada com a função de armazenar informações sobre os membros do grupo Multicast, bem como dados recebidos recentemente de cada um dos membros. A *cache* mantém uma entrada para cada membro do grupo com informações sobre este, por exemplo o endereço IP e a identificação do processo (PID). A Figura 4.2 representa uma entrada da *cache*.

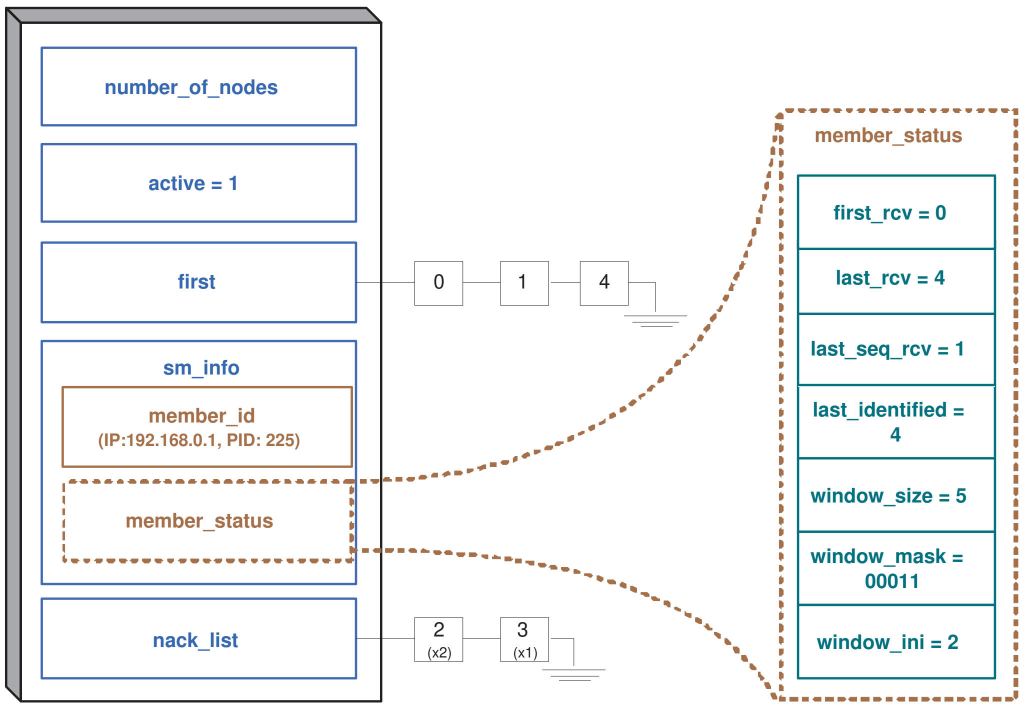

Figura 4.2: A *cache*

Cada entrada da *cache* possui os campos descritos em seguida:

- ***number_of_pckts***: número de pacotes de dados recebidos do membro e que se encontram armazenados;
- ***active***: indica se o membro está ativo ou desativado no grupo. Se *active* = 1, então o membro ainda está no grupo, senão ele já saiu do grupo;
- ***first***: ponteiro para o primeiro pacote de dados da lista dos dados recebidos. Esta lista de dados mantém somente os pacotes recebidos recentemente do membro;

- ***sm_info***: estrutura composta pelos campos: *member_id* e *member_status*;
  - *member_id*: estrutura de identificação do membro, composta pelo endereço IP do membro e a identificação do processo (PID);
  - *member_status*: estrutura que armazena o estado atual do membro. Estas informações são importantes para o processo de recuperação dos dados perdidos. A estrutura *member_status* é descrita a seguir:
    * *first_rcv*: *sn* do primeiro pacote de dados recebido do membro;
    * *last_rcv*: *sn* do último pacote de dados recebido do membro;
    * *last_seq_rcv*: *sn* do último pacote de dados em sequência recebido do membro;
    * *last_identified*: o maior *sn* armazenado na lista de dados;
    * *window_size*: número máximo de requisições de retransmissões que podem ser enviadas em um determinado instante;
    * *window_mask*: vetor com as identificações dos pacotes de dados perdidos. As informações armazenadas por este vetor são enviadas no pacote NACK. O valor 1 significa que o dado, referente à posição com o valor, foi perdido e consequentemente deve ser enviado um NACK para este dado; e o valor 2 significa que já foi enviado um NACK para o dado perdido, portanto, esperando pela retransmissão;
    * *window_ini*: posição do menor *sn* armazenado no vetor *window_mask*.
- ***nack_list***: lista que controla o número de NACKs enviados para cada pacote perdido.

Uma questão importante a ser analisada é o tamanho da *cache*. Para o protocolo Enhanced-Receiver-Initiated, se a *cache* for pequena um dado pode ser removido indevidamente caso ainda seja necessário para envio de retransmissões. A escolha do tamanho adequado para a *cache* deve levar em consideração o processo de entrada de um novo membro no grupo, caso seja feita sem a recuperação do estado atual, baseando-se no primeiro processo de entrada de membros descrito na Seção 4.3. Porém quando o processo de entrada de um novo membro no grupo baseia-se no processo ilustrado na Figura 4.3, torna-se viável a implementação de uma *cache* relativamente pequena, pois os dados que já foram removidos da *cache* no momento da entrada de um novo membro são recuperados através do recebimento do estado atual, de acordo com as explicações na Seção 4.3.

Um novo dado somente pode ser inserido na *cache* se seu armazenamento não substituir um dado da *cache* que ainda não foi enviado para aplicação. Se o dado a ser substituído na *cache* não tenha sido entregue para aplicação, então, o novo dado deve ser descartado.

O usuário da biblioteca RML pode configurar o seu funcionamento com o uso de um arquivo de configuração, definindo o tamanho da *cache* grande o suficiente para manter um bom desempenho na recuperação de dados. O arquivo de configuração permite assim adaptações do tamanho da *cache* de acordo com o tipo de aplicação que a está utilizando. As especificações técnicas sobre o uso de arquivo de configuração, assim como outras funcionalidades da biblioteca, podem ser encontradas em [34], documentação disponibilizada junto com a distribuição da biblioteca RML [1].

4.3 Gerenciamento das Sessões Multicast

Esta seção tem o objetivo de explicar o processo de entrada e de saída dos membros em um grupo Multicast. Primeiramente, é apresentado o gerenciamento de entrada de novos membros no grupo, representado pela Figura 4.3. Em seguida na Figura 4.4 é apresentado o processo de saída de um membro do grupo Multicast.

Para a garantia da confiabilidade da consistência dos dados, seria ideal que todos os membros, que pretendessem participar de uma sessão Multicast, entrassem no grupo ao mesmo tempo. Infelizmente, nem sempre isso pode ser garantido. Se algum membro entrar na sessão muito tempo após o seu início, pode não ser mais possível para este membro chegar a um estado consistente em relação ao resto do grupo. Este problema é consequência da quantidade finita de dados que podem ser armazenados na *cache* para posterior retransmissão.

Durante uma sessão Multicast, cada membro armazena uma certa quantidade de dados em uma *cache*. À medida que novos pacotes de dados vão sendo recebidos por um membro, estes são armazenados na *cache*. Como a *cache* é finita, os dados mais antigos começam a ser substituídos por dados mais novos. Portanto, se um novo membro entra no grupo no momento em que dados foram substituídos da *cache*, pode não ser mais possível recuperar os dados antigos, visto que estes já foram removidos da *cache*.

A situação apresentada acima pode não ser um problema para aplicações onde informações trocadas anteriormente não são relevantes para a consistência do estado atual, por exemplo um *chat* Multicast. Porém, se considerarmos aplicações em que existe uma dependência entre os dados, ou seja, o estado atual da sessão Multicast depende das informações passadas, esta situação deve ser tratada com muita atenção. Em uma aplicação com desenhos, como o TGWB [14], o recebimento de todos os pacotes pode ser indispensável para a consistência do estado atual. Vamos supor que o acontecimento das três ações a seguir ocorresse na ordem em que são descritas: 1) um pacote de dados referente ao desenho de um quadrado é enviado pelo membro A; 2) o membro B entra no grupo Multicast; 3) o membro A envia um pacote de dados para colorir o quadrado enviado anteriormente. Se no momento que o membro B entrar no grupo, o membro A já tiver removido os dados referentes ao desenho do quadrado de sua *cache*, o membro B não poderá recuperá-lo para colorí-lo, pois o membro A não o terá mais armazenado para retransmití-lo. Nesta situação, o membro B se encontrará em um estado inconsistente, visto que não faz sentido colorir um desenho inexistente. Entretanto, o membro pode entrar no grupo e recuperar o estado atual de algum membro antigo do grupo. Este procedimento de recuperação do estado atual será explicado a seguir.

A biblioteca RML foi implementada possibilitando a escolha da forma de entrada de um novo membro no grupo Multicast. Esta escolha está extremamente ligada ao tipo de aplicação Multicast em questão. Existem duas possibilidades para um novo membro entrar em um grupo Multicast:

1. entrar direto no grupo, sem recuperar o estado atual da sessão. Ou seja,

o membro começará a interação com o grupo sem nenhuma informação do passado. Seu estado se constituirá de tudo que ele recebeu desde o momento em que entrou no grupo Multicast;

2. recuperar o estado atual da sessão e depois entrar no grupo Multicast.

O estado atual consiste de todos os dados necessários para se tornar um membro Multicast, podendo incluir as informações armazenadas na *cache*. O estado atual varia de acordo com a aplicação, no caso do TGWB, o estado atual consiste da *cache* e dos dados contidos na tela no momento do recebimento do pedido de inclusão no grupo.

A Figura 4.3 representa a segunda possibilidade de entrada de novos membros no grupo. O candidato a novo membro envia uma mensagem *join* ao grupo Multicast e fica aguardando por uma mensagem *accept* de algum membro do grupo. Qualquer membro do grupo que receba uma mensagem *join* está capacitado para enviar uma mensagem *accept*. A mensagem *accept* contém informações necessárias (endereço e porta) para o estabelecimento de uma conexão TCP. Com o recebimento da mensagem *accept*, o candidato a novo membro estabelece uma conexão TCP com o membro emissor desta mensagem e através desta conexão será possível o envio do estado atual do antigo para o candidato a novo membro.

Quando o candidato a novo membro recebe o estado atual do antigo membro, ele pode ser considerado um membro Multicast efetivo do grupo, assim como os outros. Porém, se o tempo determinado para espera por uma mensagem *accept* expirar, o candidato a novo membro considera que não existe mais nenhum outro membro no grupo, ou seja, ele é o primeiro e único no grupo. Em ambos os casos, ele torna-se um "servidor de estado atual", isto é, está sempre preparado para enviar o seu estado atual caso receba uma mensagem *join* de algum membro que almeja entrar no grupo Multicast.

Um "servidor de estado atual" espera por conexões em uma determinada porta pré-estabelecida. Quando um membro qualquer se conecta a esta porta, o "servidor de estado atual" transmite o estado atual da aplicação para o membro requisitante.

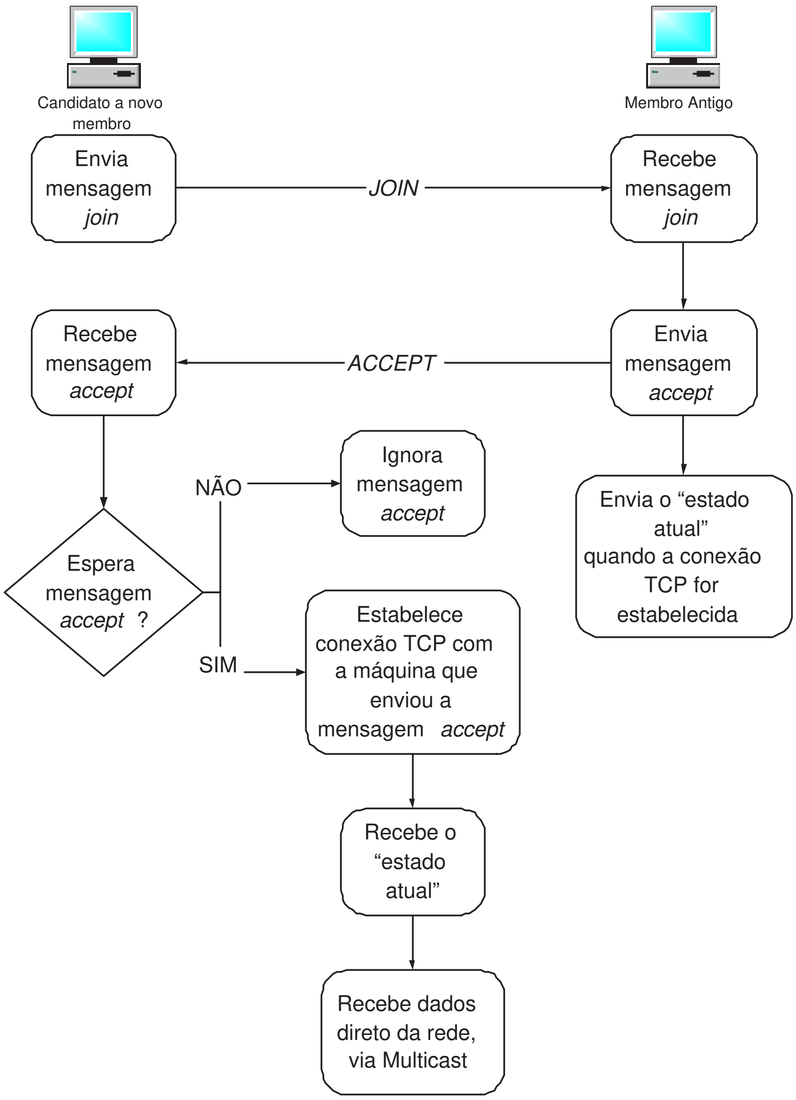

Figura 4.3: Entrada de membro no grupo Multicast

No caso da aplicação TGWB, a *cache* pertence ao estado atual da sessão, pois além de conter os dados recentemente trocados durante uma sessão, ela contém informações sobre os outros membros do grupo Multicast. Supondo que a *cache* não fosse enviada para o candidato a novo membro, e que após o recebimento do estado atual (apenas a tela) pelo candidato a novo membro, este recebesse um pacote, por exemplo com o $sn = 10$. O membro identificaria este pacote como o primeiro recebido, pois mesmo que os pacotes com *sns* de 0 a 9 tivessem sido recebidos no momento da transmissão do estado atual (tela), o novo membro acharia que tinha perdido estes pacotes, uma vez que não se encontrariam armazenados na *cache* (inicialmente a *cache* encontra-se vazia). Com isso, o membro associa esta ausência dos pacotes na *cache* como perdas e escalona NACKs referente a estes pacotes para requisitar retransmissões. Neste caso, teríamos algumas retransmissões desnecessárias, pois na verdade os pacotes de dados requisitados já foram recebidos, anteriormente, no momento do recebimento do estado atual (tela).

Quando um membro deseja sair do grupo Multicast, ele envia uma mensagem *leave group* para o grupo avisando a sua intenção e escalona sua saída para o tempo aleatório T_{LEAVE}, ou seja, escalona o evento LEV_GRP para ser disparado no tempo T_{LEAVE}. Porém, antes do envio da mensagem *leave group*, o membro envia uma mensagem *refresh*, mensagem de controle com o valor do *sn* do último pacote de dados enviado pelo membro. Caso algum membro ainda não tenha recebido os pacotes cujos *sns* são inferiores ou igual ao valor do *sn* recebido no *refresh*, serão escalonados NACKs requisitando retransmissões. Caso a mensagem *refresh* se perca e algum membro não tenha recebido todos os pacotes de dados enviados por este membro que deseja sair do grupo, então não será mais possível a recuperação de tais pacotes. Ao receberem a mensagem *leave group*, os demais membros do grupo sinalizam na *cache* que o membro que deseja sair não está mais ativo no grupo. Esta sinalização serve para cálculo dos temporizadores dinâmicos, onde é necessário calcular o atraso em relação a cada membro do grupo de tempos em tempos, com a sinalização, não será preciso calcular o atraso em relação aos membros que não estão mais ativos no grupo.

Como representado na Figura 4.4, após o término do tempo T_{LEAVE}, o evento

LEV_GRP é executado. O membro libera alguns recursos do sistema que foram alocados durante a sessão e sai do grupo Multicast.

Figura 4.4: Saída de membro do grupo Multicast

4.4 Detecção de Perdas

O processo para alcançar uma transmissão Multicast confiável pode ser dividido em duas etapas:

1. detectar as perdas dos pacotes perdidos;
2. recuperar eficientemente os pacotes detectados como perdidos.

Esta seção apresenta como é feita a detecção de perdas e na seção 4.5 é apresentado o procedimento de recuperação de dados.

A detecção de perdas pode ser feita de duas formas: a primeira através do recebimento de um pacote de dados fora de sequência. Por exemplo, um membro recebe o pacote de dados com o sn = i - 1 e, em seguida, o pacote com o sn = i + 1. Logo, é detectada a perda do pacote com o sn = i, sendo então necessária a execução dos procedimentos de recuperação dos pacotes perdidos.

A Figura 4.5 ilustra como é feito o tratamento dos dados que são recebidos durante uma sessão Multicast. Quando um membro recebe um pacote de dados, ele identifica o membro emissor e o sn do pacote recebido, verificando a existência de algum evento escalonado para este pacote recebido. Por exemplo, um pedido de retransmissão pode ter sido escalonado no caso de uma perda ter sido detectada para este pacote, porém com o recebimento do pacote não é mais necessário o pedido de retransmissão. Se o membro emissor, identificado no pacote, ainda não possui uma entrada na *cache*, então é feita sua inclusão. O pacote de dados recebido, que ainda não se encontra na *cache* e que seu armazenamento não substitua um dado ainda não enviado para aplicação, deve ser armazenado na *cache*. Se o sn do pacote de dados recebido for o esperado, o dado é então passado para a aplicação. Porém, se este pacote de dados não for o esperado, ou seja, se o sn for maior que o sn anteriormente recebido + 1, então o dado somente será liberado para a aplicação, após a recuperação de todos os pacotes de dados perdidos com os sns inferiores ao pacote de dados recebido. Deste modo, os pacotes são entregues ordenados para a aplicação.

Caso haja detecção de perda devido ao recebimento de um pacote de dados com o sn não esperado, é feita a verificação da existência de eventos escalonados para os pacotes que foram detectados como perdidos. Primeiro, é verificada a existência do evento para envio do NACK (evento NACK_SND), depois a existência do evento para espera por retransmissão antes do reenvio do NACK (evento RET_RCV), se após estas verificações for identificado que não existe nenhum evento escalonado para os pacotes perdidos, então é escalonado o evento NACK_SND, ou seja, o evento é incluído na lista de eventos (maiores detalhes sobre o funcionamento da lista de eventos encontram-se na Seção 4.6). Se já existir o evento NACK_SND ou RET_RCV escalonado na lista de eventos, então, nada deve ser feito, pois o procedimento de recuperação dos dados já foi iniciado.

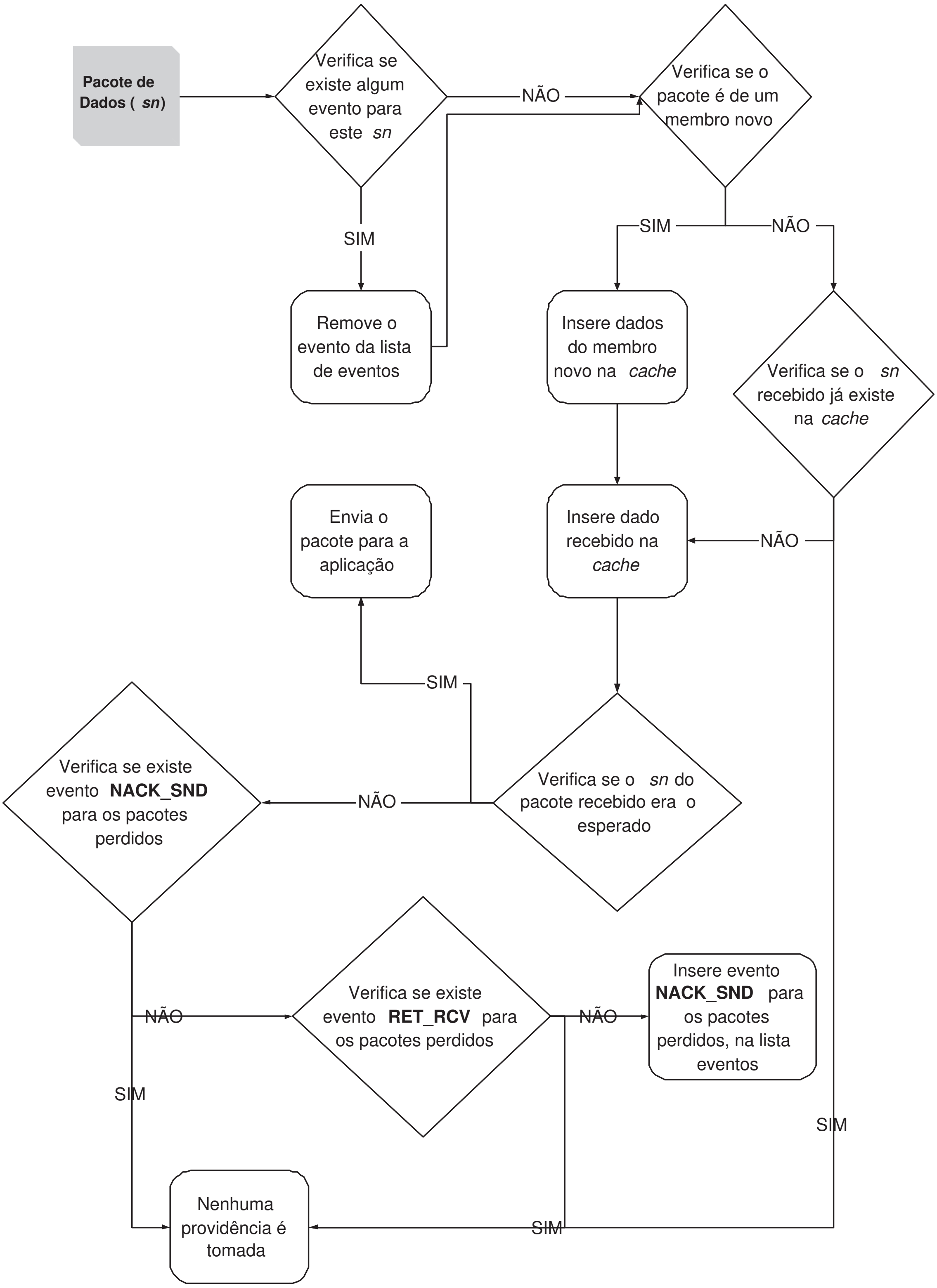

Figura 4.5: Tratamento de dados

A segunda forma de detecção de perdas é feita através do recebimento das mensagens *refresh*. A mensagem *refresh* contém a identificação do membro que a emitiu e o valor do *sn* do último pacote de dados por ele enviado. Na atual implementação da biblioteca, as mensagens *refresh* são enviadas cada 10s. Quando o membro recebe mensagens *refresh* e verifica que pacotes com *sns* inferiores ou igual ao valor do *sn* recebido no *refresh* não foram recebidos, ele detectada perdas. De maneira similar ao tratamento de dados, após a detecção das perdas, é feita a verificação da existência de eventos para estes pacotes perdidos. Se já existir o evento NACK_SND ou RET_RCV escalonado na lista de eventos, então, nada deve ser feito, pois o procedimento de recuperação dos dados já foi iniciado. Porém, não havendo existência de eventos escalonados, então o evento NACK_SND é escalonado. O tratamento da mensagem *refresh* está representado na Figura 4.6.

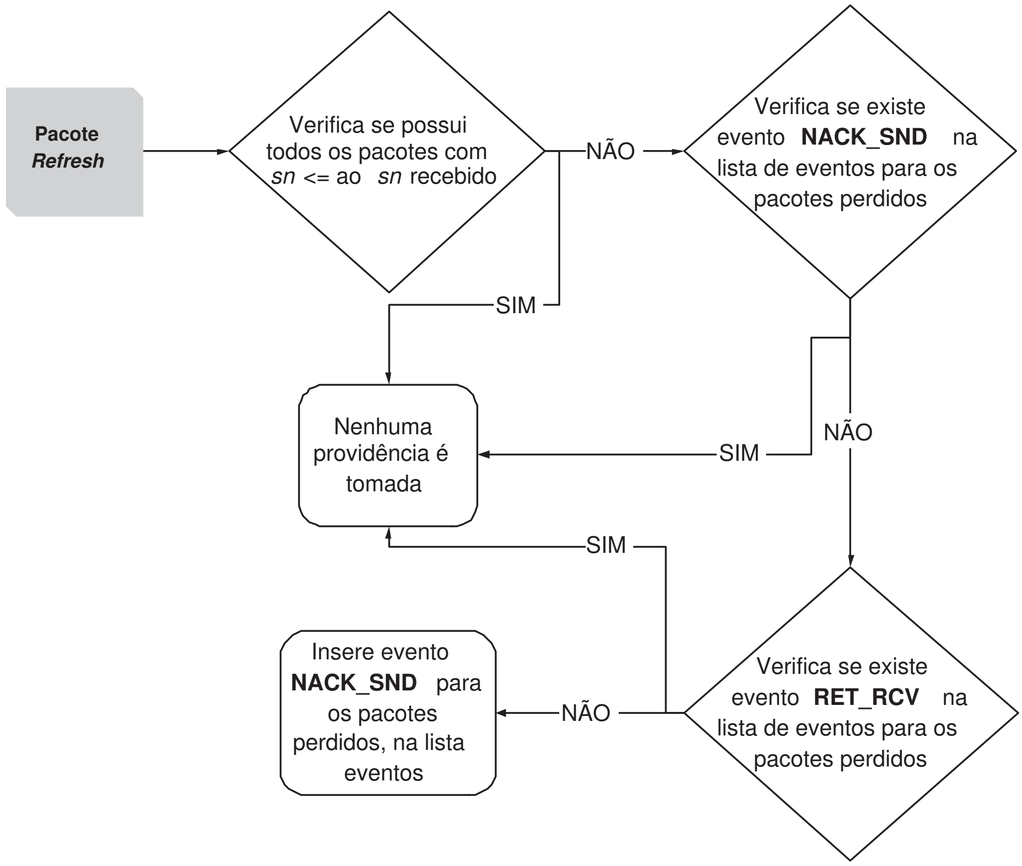

Figura 4.6: Tratamento da mensagem *refresh*

O uso da mensagem *refresh* é útil em situações onde se perde o último pacote de dados enviado por um determinado membro. Nesta situação, os membros receptores não teriam como identificar a perda, utilizando somente a primeira forma de detecção de perdas descrita nesta seção. Suponha que um membro envie o pacote de dados com o $sn =$ i e que algum membro tenha perdido este pacote, se este pacote perdido for o último pacote enviado pelo membro emissor, então, a única forma de recuperá-lo é com o recebimento da mensagem *refresh*.

4.5 Recuperação de Dados

O procedimento de recuperação consiste, primeiramente, na solicitação das retransmissões dos pacotes perdidos, seguido do aguardo pelo recebimento das retransmissões solicitadas e finalizando no recebimento destas retransmissões.

4.5.1 Envio de NACKs

No mecanismo de envio de NACKs foi implementado o uso de temporizadores para evitar que ocorra a implosão de NACKs. Como descrito na apresentação do protocolo Enhanced-Received-Initiated no Capítulo 2, a implosão de NACKs é a grande quantidade de NACKs iguais trafegando pela rede. Isso acontece quando dois ou mais membros do grupo Multicast enviam NACKs solicitando retransmissões para os mesmos pacotes de dados, podendo gerar o congestionamento da rede.

Como o simples envio de NACKs, no instante da detecção da perda, pode causar a implosão de NACKs, ao invés de enviar NACKs imediatamente após a detecção da perda, um evento NACK_SND é escalonado para acontecer no tempo aleatório T_{NACK}. Caso algum membro do grupo, que também tenha perdido o mesmo pacote, escalone o evento NACK_SND para um tempo menor que o T_{NACK} dos outros membros, então o evento deste membro pode ser disparado antes, enviando o NACK e suprimindo o envio do NACK dos demais membros do grupo. Assim, se escolhermos uma maneira eficiente para determinar T_{NACK}, teremos uma grande probabilidade de suprimir o envio de NACKs duplicados na rede.

O tratamento da mensagem NACK pode ser representado pela Figura 4.7, onde a cada recebimento de uma mensagem NACK é feita a verificação da existência do evento NACK_SND, relativo ao mesmo grupo de pacotes, na lista de eventos. Caso exista, o evento é removido, suprimindo o NACK que estava escalonado para ser enviado e escalonando o evento RET_RCV para esperar pela retransmissão durante o tempo aleatório T_{WAIT}. Caso não exista o evento NACK_SND escalonado para este(s) pacote(s), então é verificada a existência do evento RET_RCV. Caso exista, significa que o membro receptor do NACK já enviou um NACK para o pacote em questão e está aguardando a retransmissão do pacote perdido. Porém, se não existir o evento RET_RCV, o membro pode agir de duas maneiras: se o membro não tiver os pacotes requisitados na *cache*, então, o evento RET_RCV deve ser escalonado para esperar por estas retransmissões já requisitadas; se o membro tiver os pacotes requisitados na *cache*, então, o evento RET_SND deve ser escalonado para envio das retransmissões.

Após o envio do NACK, o membro espera pela retransmissão durante o tempo aleatório T_{WAIT}. Ao fim, o membro escalona um novo NACK referente os pacotes que ainda não foram recuperados. Essa situação pode se repetir algumas vezes, dependendo do número máximo de NACKs que pode ser enviado para o mesmo pacote perdido. O usuário da biblioteca RML pode definir este número no arquivo de configuração, como descrito na documentação [34]. O campo *nack_list* da *cache* (Figura 4.2) armazena quais e quantos NACKs foram enviados. Se for enviado o número máximo de NACKs permitido para o mesmo pacote, entende-se que não foi possível recuperar o pacote perdido e a aplicação deve ser finalizada.

A eficiência do mecanismo de envio de NACKs envolve outros fatores, como por exemplo o controle do número de requisições contidas em um pacote NACK. O fato de um membro requisitar mais retransmissões de dados do que ele é fisicamente capaz de receber, pode aumentar desnecessariamente o tráfego na rede. Portanto, é necessário que se faça o controle de fluxo do número de requisições. Supondo uma *cache* com a capacidade de armazenamento de 50 pacotes e que um membro

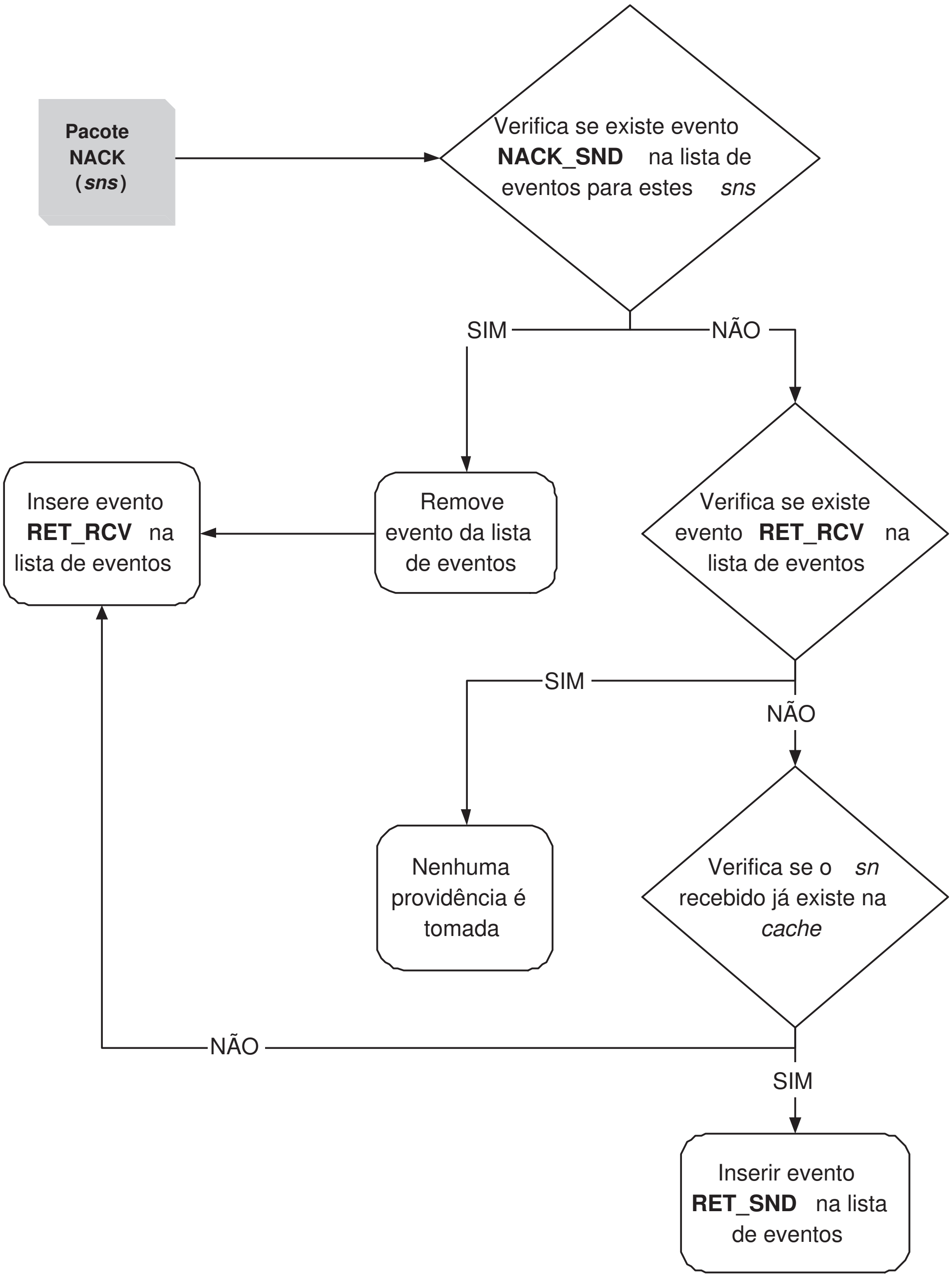

Figura 4.7: Tratamento da mensagem NACK

tenha detectado a perda de 100 pacotes de dados: se não houver controle do número de pacotes a serem requisitados, o membro pode pedir retransmissões de todos os pacotes perdidos ao mesmo tempo, havendo retransmissões de pacotes que momentaneamente não devem ser inseridos na *cache*, pois substituiriam pacotes que ainda não foram enviados para a aplicação.

Para evitar o envio de retransmissões desnecessárias, não devem ser requisitados mais pacotes do que a *cache* é capaz de armazenar. Quando muitos pacotes forem perdidos, é enviado apenas um NACK requisitando a retransmissão de no máximo w pacotes com os *sns* em sequência, onde w é o número máximo de requisições de retransmissão que um membro pode enviar ao mesmo tempo, sendo w menor ou igual ao tamanho da *cache*. Por questões de implementação, o número escolhido para w foi 64, este valor representa dois inteiros que podem ser passados no pacote NACK ao invés de 64 valores de *sns*. Os dois inteiros podem ser interpretados através da máscara apresentada na Figura 4.8. O pacote NACK além de conter a identificação do membro que o emitiu e do membro emissor do dado perdido, contém os seguintes campos:

- sn_base : sn da primeira requisição da máscara;
- w : número máximo de retransmissões que podem ser solicitados ao mesmo tempo;
- $hmask$: o inteiro que representa a parte alta da máscara;
- $lmask$: o inteiro que representa a parte baixa da máscara.

O cálculo dos *sns* requisitados no NACK é feito a partir da transformação dos números inteiros contidos em *hmask* e *lmask* em sua representação binária. Todas as posições com o valor 1 significam um pedido de retransmissão e o valor deste sn pode ser calculado através da soma da posição em que se encontra com o valor do sn_base. Supondo que um membro receba um NACK com o $sn\_base = 5$, $w = 64$, $hmask = 1$ e $lmask = 3$, como mostra a Figura 4.8, os *sns* dos pacotes requisitados seriam 5 $(0 + 5)$, 6 $(1 + 5)$ e 37 $(32 + 5)$.

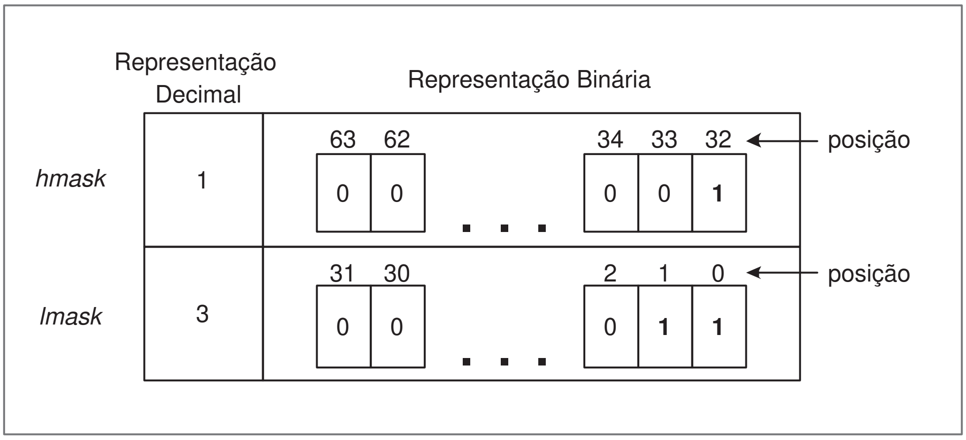

Figura 4.8: Máscara de NACKs

A Figura 4.9 representa dois mecanismos que podem ser usados para o envio do NACK. O primeiro mecanismo não limita o número de requisições feitas em um determinado tempo. Ele envia um NACK para cada pacote perdido e armazena na *cache* todas as retransmissões recebidas. O segundo, usado pela biblioteca RML, mantém o controle do número máximo de requisições enviadas e envia apenas um NACK contendo todas as requisições de retransmissões que podem ser solicitadas. Para análise da Figura 4.9, vamos supor o seguinte cenário: uma *cache* com a capacidade de armazenamento para apenas 5 pacotes; um membro emissor enviando pacotes de dados com *sn* do 0 até 8, que todos os pacotes são perdidos pelo membro receptor; e que no envio das retransmissões é perdido o pacote com o $sn = 0$. O recebimento do *refresh* pelo receptor o torna capaz de detectar a perda e requisitar a retransmissão através do envio do(s) NACK(s).

No primeiro mecanismo da Figura 4.9, o receptor envia nove NACKs na rede, um para cada pacote perdido. O armazenamento das retransmissões dos pacotes com os *sns* 6, 7 e 8 faz com que os pacotes com os *sns* 1, 2 e 3 sejam substituídos da *cache*. Porém, como foi suposta a perda da retransmissão do pacote com o $sn = 0$, estes ainda não haviam sido entregues para a aplicação, uma vez que os pacotes devem ser entregues em ordem para a mesma. Portanto, é necessário o reenvio de NACKs, causando o aumento do tempo de recebimento dos dados e do tráfego na rede.

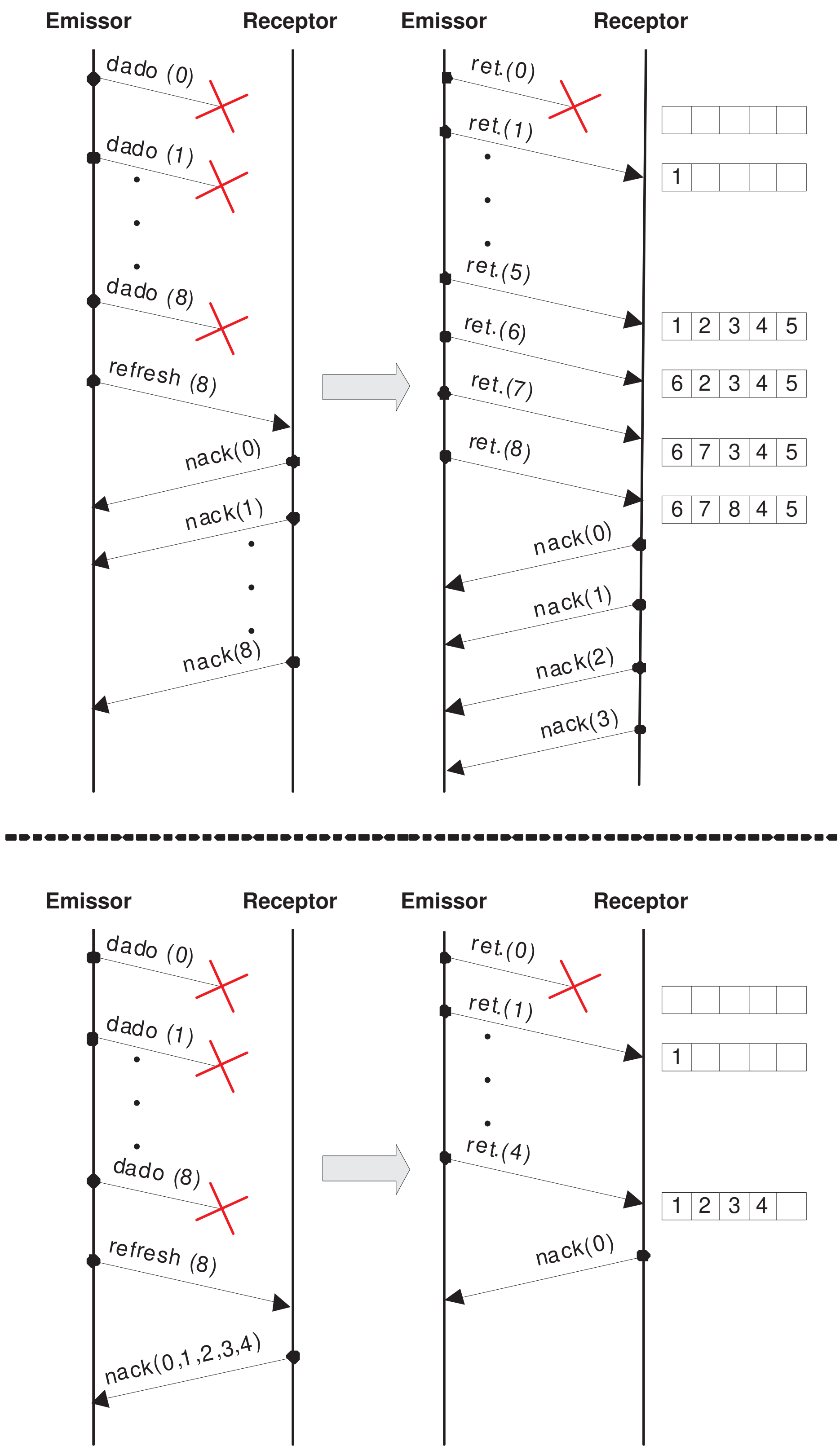

Figura 4.9: Mecanismos de controle no envio do NACK

No segundo mecanismo, apenas um NACK é enviado e o número máximo de requisições feitas neste NACK é igual ao tamanho da *cache*. Não havendo retransmissões desnecessárias, consequentemente não será necessário o envio de NACK para pacotes de dados já recebidos antes. Note-se que apenas o pacote com o $sn = 0$ é requisitado no segundo NACK enviado. Isto acontece porque neste momento requisições de retransmissões para os pacotes com sns superiores a 4 ainda não devem ser inseridos na cache, pois substituiriam dados ainda não entregues a aplicação. Com o emprego do segundo mecanismo, podemos perceber que o número de pacotes trafegando pela rede pode ser consideravelmente reduzido.

4.5.2 Retransmissão de Dados

A biblioteca RML considera que cada membro mantém em sua *cache* os últimos n pacotes de dados enviados durante a sessão Multicast. Sendo assim, todos os membros estão aptos a retransmitirem qualquer um dos n pacotes armazenados em sua *cache*, ou seja, a responsabilidade da retransmissão é distribuída entre todos os membros da sessão Multicast. Com isso, pode acontecer que retransmissões duplicadas sejam enviadas pela rede. Para evitar este problema, foi implementado o uso de temporizadores similar ao utilizado no envio dos NACKs.

Se o membro A receber um NACK requisitando a retransmissão de um pacote de dados que está contido em sua *cache*, então deverá escalonar o evento RET_SND para ser disparado no tempo aleatório T_{RET}. Caso este membro receba uma retransmissão do pacote requisitado, antes do tempo T_{RET} expirar, ele considera que algum outro membro do grupo já fez a retransmissão que pretendia fazer e cancela o envio da retransmissão que estava escalonada para ser enviada. Porém, se nenhuma retransmissão chegar até o tempo T_{RET} expirar, o membro A envia a retransmissão solicitada para o grupo. Caso o dado a ser retransmitido seja removido da *cache* antes do disparo do evento RET_SND, quando o membro ao busca o dado para retransmití-lo, detecta a sua remoção da *cache* e, consequentemente, não retransmite o pacote solicitado.

Quando um membro do grupo Multicast recebe uma retransmissão, ele verifica a existência do evento RET_SND referente à retransmissão recebida. Caso exista, o evento é cancelado, conforme explicado no parágrafo anterior. Contudo, se não houver o evento RET_SND escalonado, é feita a verificação da existência dos eventos NACK_SND e RET_RCV, ou seja, o membro verifica se ele já havia detectado a perda desta retransmissão recebida e se pretendia ou se já havia mandado um NACK para este pacote. Caso algum dos dois eventos tenha sido escalonado, então, o evento escalonado é cancelado e o dado retransmitido é armazenado na *cache*. A Figura 4.10 demonstra o tratamento da mensagem de retransmissão.

4.6 Lista de Eventos

Todas as ações tomadas pela biblioteca RML são escalonadas para algum tempo no futuro, ou seja, nenhuma ação é feita imediatamente após a detecção de sua necessidade. Os eventos, escalonados para acontecerem no futuro, são inseridos em uma lista, chamada lista de eventos, e permanecerão nesta lista durante o tempo determinado pelo temporizador. Quando o tempo expira o evento é removido da lista e executado. Porém, um evento pode ser cancelado da lista antes do temporizador expirar.

Escalonar eventos é uma atividade extremamente comum da RML. Por exemplo, espera para envio de um NACK, pelo escalonamento do evento NACK_SND, ou seja, pela inclusão deste evento na lista de eventos para acontecer no tempo T_{NACK}. O mesmo acontece quando se espera por uma retransmissão, para enviar uma retransmissão ou para sair da sessão Multicast.

Quando o temporizador para um determinado evento dispara, ele é executado. Sua execução vai depender do seu tipo. A seguir serão descritos os tipos de eventos e as respectivas ações tomadas quando disparados:

NACK_SND : o NACK é enviado e o evento RET_WAIT é inserido na lista de eventos para esperar pela retransmissão do pacote requisitado.

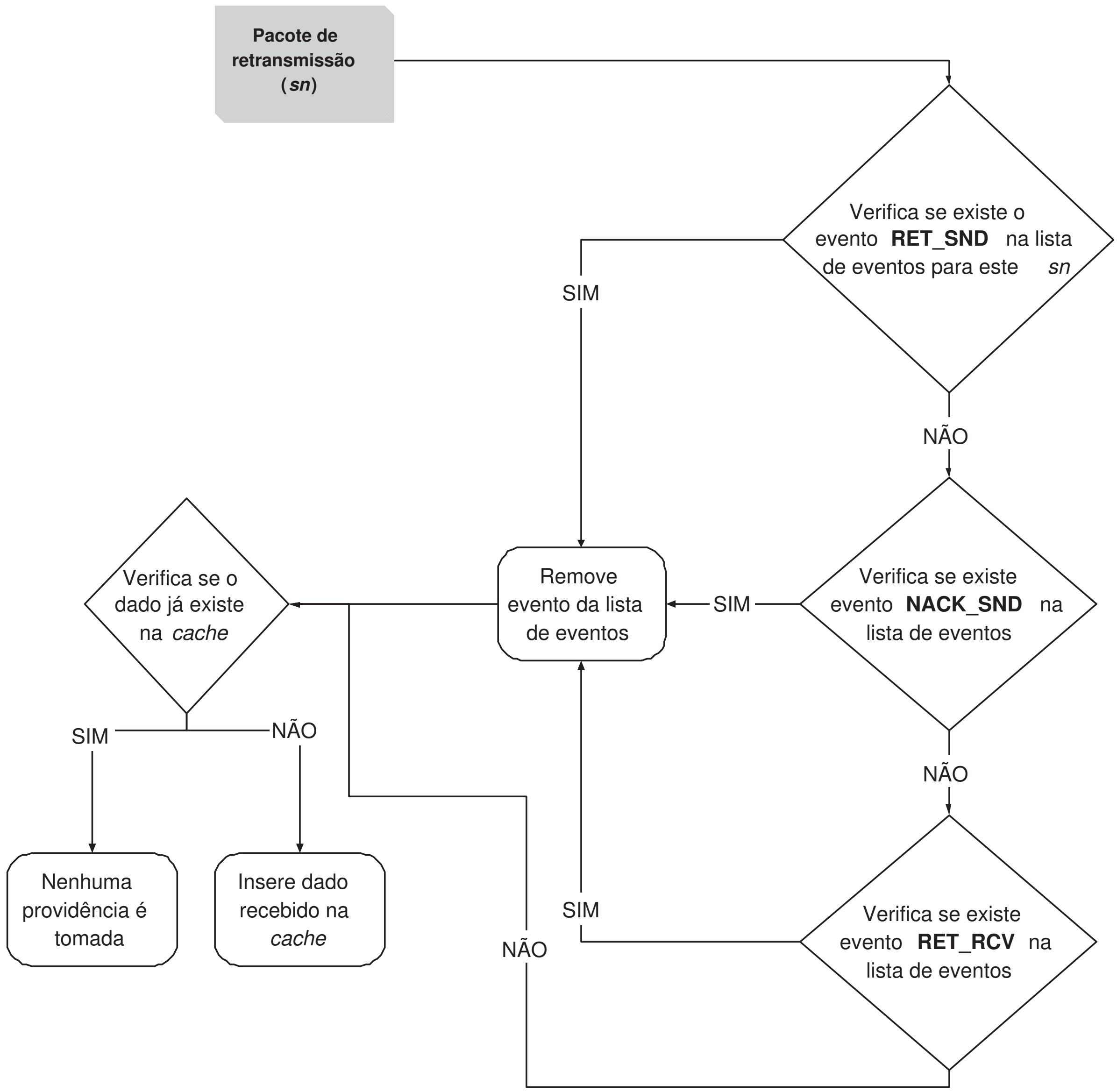

Figura 4.10: Tratamento da retransmissão

RET_WAIT : o evento NACK_SND é inserido na lista de eventos para reenvio do NACK.

RET_SND : a retransmissão é enviada.

LEV_GRP : o membro libera alguns recursos do sistema que foram alocados durante a sessão Multicast e sai do grupo Multicast.

REF_SND : a mensagem *refresh* é enviada, identificando o *sn* do último pacote de dados enviado.

Em algumas situações, o evento pode ser removido da lista de eventos antes do temporizador expirar. São elas:

- se o membro receber um NACK requisitando a retransmissão do mesmo pacote de dados que este deseja requisitar. Neste caso não é mais necessário o envio do NACK, sendo o evento NACK_SND removido da lista de eventos;
- se o membro receber uma retransmissão e existir o evento NACK_SND ou o evento RET_WAIT na lista de eventos, o evento deve ser removido pois a retransmissão desejada já foi recebida;
- se o membro receber uma retransmissão e existir o evento RET_SND, o evento deve ser removido pois algum outro membro já fez a retransmissão que estava escalonada para ser feita.

A lista de eventos armazena os eventos a serem executados, organizados em ordem crescente de acontecimento. Cada nó da lista de eventos contém as informações necessárias para execução do evento, como o tipo do evento e o tempo de execução para o qual foi escalonado. O tempo de execução do evento armazenado em cada nó é relativo ao tempo de execução do nó que o antecede na lista de eventos. Desta forma apenas 1 temporizador é necessário para implementar as ações.

A Figura 4.11 representa um exemplo simplificado da lista de eventos, onde é suposta a existência de cinco eventos na lista, escalonados para 4, 6, 6, 13 e 17 unidades de tempo no futuro. O tipo e os outros atributos do evento foram omitidos para fins de simplificação. Os valores contidos nos nós representam o tempo relativo de disparo de cada evento da lista. Como o segundo e o terceiro evento foram escalonados para serem executados no tempo 6, o terceiro nó contém o tempo relativo igual a 0, ou seja, o terceiro evento deve ser executado 0 unidades de tempo após a execução do segundo evento.

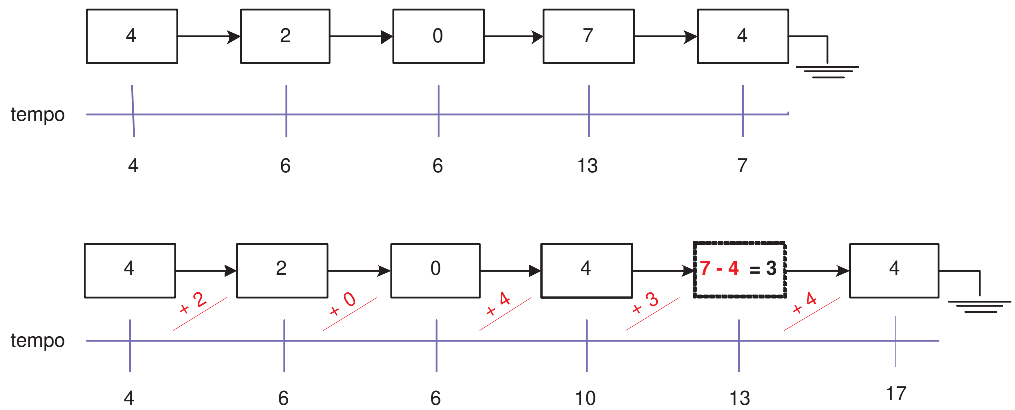

Figura 4.11: Lista de eventos

Toda vez que um evento é inserido no início da lista, o sinal de alarme do sistema operacional é escalonado para disparar no tempo indicado pelo nó inicial da lista de eventos. Quando o alarme dispara, o evento é removido da lista e processado. Todos os nós subsequentes que tiverem seu tempo relativo igual a 0, deverão ser removidos e processados imediatamente. O alarme é então reiniciado para disparo no tempo do novo nó inicial da lista.

Como os tempos de disparo são relativos, devem ser atualizados toda vez que um evento for inserido ou removido da lista. A Figura 4.11 demonstra a inserção de um evento escalonado para o tempo 10. Este evento deve ocorrer após o evento escalonado para o tempo 6 e antes do evento escalonado para o tempo 13. O tempo relativo do evento escalonado para o tempo 13 deve ser atualizado de 7 para 3, pois agora este evento vai ocorrer 3 unidades de tempo após o evento que o antecede. A remoção de um evento acontece de maneira similar à inserção, porém o evento sucessor ao evento removido deve ter seu tempo relativo somado ao tempo relativo do evento que está sendo removido.

4.7 Temporizadores

O objetivo da implementação de um algoritmo para cálculo dos temporizadores é a supressão de pacotes duplicados na rede e a otimização do tempo de recuperação dos pacotes perdidos em uma sessão Multicast. Existem alguns estudos sobre o assunto na literatura [21, 30, 31].

A pesquisa feita em [30] tem o objetivo de estudar a relação entre o cálculo dos temporizadores e o desempenho da recuperação de pacotes perdidos ou com erros no protocolo SRM [19]. São apresentadas fórmulas para o cálculo dos temporizadores, baseando-se nos atrasos entre os membros do grupo e rotinas para a adaptação dinâmica dos temporizadores em relação às mudanças na rede, como o número de membros vizinhos. O tamanho da vizinhança é calculado dinamicamente baseado no desempenho da recuperação dos dados. A implosão de NACKs é evitada utilizando-se temporizadores aleatórios com distribuição uniforme. Segundo [30], a supressão de requisições e retransmissões não garantem a ausência de pacotes duplicados na rede, pois uma segunda requisição enviada antes do recebimento da retransmissão é também um pacote duplicado. E esta segunda requisição ainda pode causar duplicação da retransmissão, devendo as requisições prematuras e as retransmissões duplicadas serem tratadas. Esta pesquisa demonstra que é possível prevenir requisições prematuras e ignorar, dentro de um intervalo de tem pedidos duplicados de retransmissões.

No processo de ajuste dinâmico dos temporizadores (Figura 4.12), o tamanho da vizinhança é interpretado a partir das mensagens duplicadas e a partir do atraso de recuperação. Devido à média do número de requisições na vizinhança ser proporcional ao tamanho da vizinhança, se houver muitas mensagens duplicadas na rede significa que o tamanho da vizinhança foi subestimado e que deve ser aumentado. Porém, não é possível determinar com precisão quando o tamanho da vizinhança foi superestimado e deve ser reduzido, pois o recebimento de poucas mensagens duplicadas pode ser consequência de um cálculo otimizado para os temporizadores, não necessitando do seu ajuste.

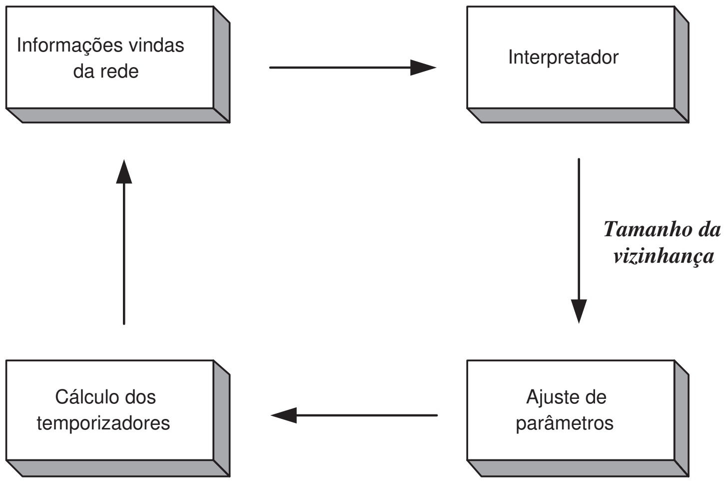

Figura 4.12: Fluxo de controle do ajuste dinâmico dos temporizadores

O cálculo do atraso é inversamente proporcional ao tamanho da vizinhança, ou seja, o aumento da vizinhança gera a diminuição do atraso uma vez que o tempo de espera para envio de uma requisição pode ser menor devido à grande quantidade de vizinhos que o membro possui. Portanto, se o atraso diminui, significa que a vizinhança aumentou. Em [30] é demonstrado que o ajuste do tamanho da vizinhança, através do atraso de recuperação, é mais eficiente do que através da quantidade de mensagens duplicadas na rede.

Um outro exemplo de estudos dos cálculos dos temporizadores é a pesquisa realizada em [31], onde um método é proposto para evitar a implosão de NAKs baseando-se em temporizadores com distribuição exponencial. É mostrado através de análises e simulações que a implosão de NACKs pode ser evitada, mantendo-se o mecanismo de supressão sempre robusto, mesmo com a perda de NACKs. O mecanismo tem uma complexidade computacional baixa em cada receptor, que é independente do tamanho do grupo, pois é considerado apenas um emissor por sessão. Cada receptor mantém armazenado o atraso apenas em relação ao emissor. Nenhum suporte nem conhecimento sobre a topologia da rede são necessários para evitar a implosão de NACKs. Os resultados das simulações mostram que a supressão de NACKs com o uso dos temporizadores, com distribuição exponencial, é escalável em relação ao número de receptores e que previne a implosão de NACKs, ao mesmo tempo que modera o atraso na recuperação dos dados perdidos. O cálculo dos temporizadores é baseado no número de receptores do grupo. Portanto o mecanismo preocupa-se com o número estimado de receptores, onde um número de receptores R superestimado aumenta o intervalo do temporizador, aumentando, consequentemente, o atraso de recuperação dos dados perdidos e um número de receptores R subestimado diminui o intervalo do temporizador, podendo causar a implosão de NACKs. As principais diferenças entre este mecanismo [31] e o utilizado pelo protocolo SRM [30] descrito anteriormente são:

- o protocolo SRM usa uma distribuição uniforme para cálculo dos temporizadores dependente do atraso entre emissor e receptor, enquanto este mecanismo usa uma distribuição exponencial baseado no número de receptores;
- o protocolo SRM previne a perda de NACKs, escalonando uma segunda requisição em um intervalo de tempo maior que o anterior;
- o protocolo SRM ignora pedidos de retransmissão repetidos durante um determinado intervalo de tempo.

Em [21] é apresentado um algoritmo distribuído para computar o cálculo determinístico dos temporizadores, baseado na topologia da rede e no atraso entre o emissor e o receptor. É considerado apenas um emissor por sessão e os receptores não mantêm uma *cache* para armazenamento dos dados trocados durante uma sessão. Esta função compete somente ao emissor, que é o único responsável pelas retransmissões para o grupo. Quando um receptor detecta uma perda, este envia um NACK, via Unicast, para o emissor. Portanto, o receptor não precisa ter conhecimento dos demais membros do grupo, apenas precisando saber o valor do atraso entre ele e o emissor.

Segundo [21], a implosão de NACKs pode ser evitada se para cada perda de dados apenas um NACK é enviado cedo, o suficiente para que a retransmissão ocasionada por este NACK suprima os NACKs escalonados pelos demais receptores do grupo. Isto é, a retransmissão deve chegar nos demais receptores antes que o tempo de espera, para envio do NACK, expire.

A biblioteca RML implementada não tem o objetivo de otimizar o cálculo dos temporizadores. Nesta dissertação é assumida uma forma simples de encontrá-los baseando-se nos atrasos. Porém, a biblioteca permite que um algoritmo mais complexo seja construído com a responsabilidade de otimizar o cálculo dos temporizadores.

Na RML, os temporizadores foram implementados com distribuição uniforme. Os temporizadores são estáticos, ou seja, são definidos no início da sessão Multicast através do arquivo de configuração e não se alteram durante a mesma. Os temporizadores T_{NACK}, intervalo de tempo que o membro espera para solicitar uma retransmissão; T_{WAIT}, intervalo de tempo que o membro espera para solicitar novamente uma retransmissão já solicitada anteriormente; e T_{RET}, intervalo de tempo que o membro espera para retransmitir um dado solicitado, foram definidos dentro dos seguintes intervalos de tempo:

$$T_{NACK} = \{\mathrm{A} * delay_{receptor,emissor}, (\mathrm{A} + \mathrm{B}) * delay_{receptor,emissor}\}$$

$$T_{WAIT} = \{\mathrm{C} * delay_{receptor,emissor}, (\mathrm{C} + \mathrm{D}) * delay_{receptor,emissor}\}$$

$$T_{RET} = \{\mathrm{E} * delay_{receptor,emissor}, (\mathrm{E} + \mathrm{F}) * delay_{receptor,emissor}\}$$

onde $delay_{receptor,emissor}$ é o atraso entre o membro receptor e o membro emissor do dado transmitido e A, B, C, D, E e F são constantes a serem escolhidas pelo usuário.

A Figura 4.13 ilustra o intervalo dos temporizadores conforme as definições apresentadas anteriormente.

Ao detectar uma perda, o membro escolhe, aleatoriamente, com distribuição uniforme, dentro do intervalo T_{NACK}, um valor para o tempo de espera antes de solicitar a retransmissão. Com esta definição, temos uma alta probabilidade de que o envio de um NACK por um membro mais próximo ao emissor suprima o envio de um NACK por um membro mais distante do emissor, já que o membro mais próximo ao emissor possui o intervalo T_{NACK} menor do que os intervalos dos membros mais distantes.

Para os exemplos a seguir é suposto que as constantes A e E possuem o valor 1 e B, C, D e F, o valor 2, e que o valor dos temporizadores seja o limite inferior

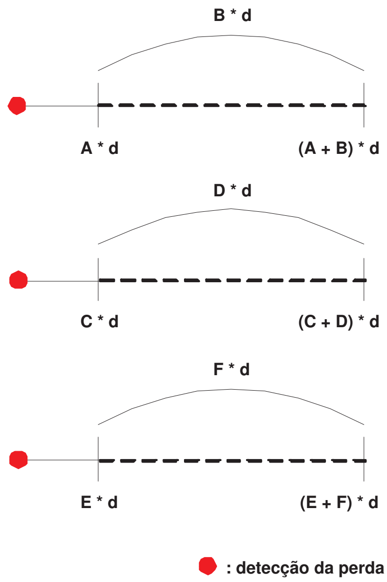

Figura 4.13: Intervalo dos temporizadores

do intervalo de cada temporizador. Suponha os atrasos representados pela Figura 4.14, sendo o membro A o emissor e os membros B e C os receptores, e que as nuvens representam os canais com os atrasos entre os membros. Suponha também que os receptores B e C tenham perdido um dado enviado pelo emissor A, e que o receptor B detecte a perda antes do receptor C por ter um atraso menor em relação ao emissor. O receptor C poderá só detectar a perda 30 u.t. (unidades de tempo) após a detecção feita pelo receptor B. A supressão de NACK pode ser observada na Figura 4.15, onde o T_{NACK} do receptor B (20 u.t.) mais o atraso entre B e C (20 u.t.) é menor que o tempo que o receptor C leva para detectar a perda do dado (30 u.t.) mais o T_{NACK} deste (50 u.t.). Logo, o receptor B envia um NACK que chega no receptor C no tempo $t = 40$ u.t., suprimindo o NACK que estava escalonado para ser enviado por este para o tempo $t = 80$ u.t..

Quando um membro recebe um NACK, ele deve calcular o tempo de espera para retransmitir o dado solicitado através da escolha aleatória dentro do intervalo T_{RET}. No caso do receptor do NACK ser o próprio emissor, ao invés de não esperar para retransmitir, uma vez que o atraso entre o emissor e o receptor é zero e ele é o emissor e o receptor ao mesmo tempo, este deve esperar um tempo aleatório que foi definido igual à média dos T_{RET} do grupo. Assim, o envio da retransmissão pelo

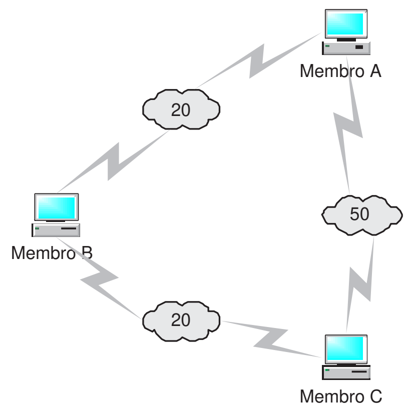

Figura 4.14: Atrasos entre membros em uma sessão Multicast

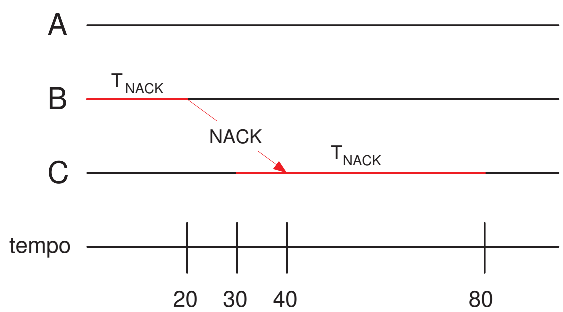

Figura 4.15: Análise da supressão de NACK

emissor pode ser suprimida em alguns casos, como demonstrado na Figura 4.16, onde também é suposto que os membros A, B e C possuem os atrasos representados pela Figura 4.14 e que o valor dos temporizadores seja o limite inferior do intervalo de cada temporizador. O receptor B recebe um NACK do receptor C no tempo t = (50 + 20) u.t., onde 50 é o tempo de espera do receptor C antes de enviar o NACK e 20 é o atraso entre C e B. O emissor A recebe o mesmo NACK no tempo t = (50 + 50) u.t., onde 50 é o tempo de espera do receptor C antes de enviar o NACK e 50 é o atraso entre C e A. Supondo o T_{RET} do emissor A igual a média dos T_{RET} do grupo, o emissor A retransmitiria o dado no tempo t = (100 + 35) u.t., onde 100 é o tempo em que o NACK chegou em A e 35 é o tempo de espera para retransmitir definido por T_{RET}. O receptor B retransmitiria o dado no tempo t = (70 + 20) u.t., onde 70 é o tempo em que o NACK chegou em B e 20 é o tempo de espera para retransmitir definido por T_{RET}. Logo, esta retransmissão chegará em A no tempo t = (90 + 20) u.t., onde 90 é o tempo que o receptor B retransmitiu o dado e 20 é o atraso para a retransmissão chegar no emissor A e anular o envio da retransmissão por este, que estava escalonada para o tempo t = (100 + 35) u.t..

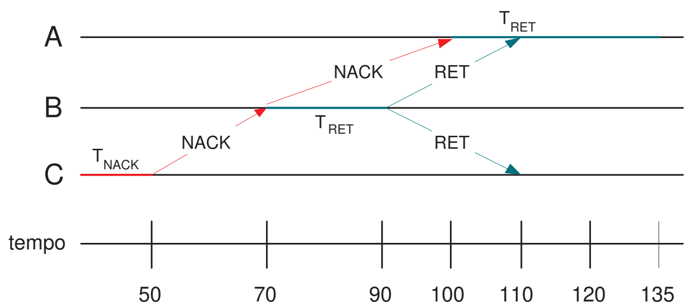

Figura 4.16: Análise da supressão de retransmissão

Supondo o caso de uma apresentação no Whiteboard TGWB (Capítulo 3), onde há somente um emissor e n receptores, cada receptor só precisa manter em sua *cache* informações sobre o emissor, mantendo transparente a comunicação entre os membros do grupo Multicast. Se o temporizador T_{RET} fosse relativo ao emissor do NACK, ao invés de ser relativo ao emissor do dado perdido, seria necessário guardar as informações deste requisitante de retransmissão na *cache*, ou seja, o receptor manteria na *cache* as informações de todos os receptores que alguma vez já lhe enviaram algum NACK, o que pode prejudicar a escalabilidade do protocolo.

A definição do temporizador T_{WAIT} foi estabelecida para que após o envio de um NACK, o receptor espere o tempo suficiente para que este NACK chegue ao emissor e que a retransmissão enviada pelo emissor seja recebida. Após este tempo expirar, o receptor ainda espera por mais o intervalo de tempo T_{NACK} antes de finalmente reenviar o NACK. Este tempo deve ser suficientemente grande para que a retransmissão chegue no receptor e suprima o reenvio do NACK. Com certeza, o emissor possui o dado requisitado e para o pior caso somente ele terá o dado, portanto o cálculo do temporizador T_{WAIT} foi baseado no atraso do receptor em relação ao emissor. Ainda considerando os atrasos da Figura 4.14 e que o valor dos temporizadores seja o limite inferior do intervalo de cada temporizador, suponha que apesar de ambos os receptores terem perdido o dado, somente o receptor C tenha detectado a perda. Logo, se o T_{NACK} do membro C for igual a 50 u.t. (atraso entre o receptor C e o emissor A) e o T_{WAIT} do membro C for igual a 100 u.t. (2 x (atraso entre o receptor C e o emissor A)), então, após enviar um NACK, o receptor deve esperar mais 150 u.t. antes do reenvio do NACK. Como pode ser observado na Figura 4.17, o reenvio do NACK pode ser suprimido se o emissor A retransmitir o dado antes do receptor C reenviá-lo.

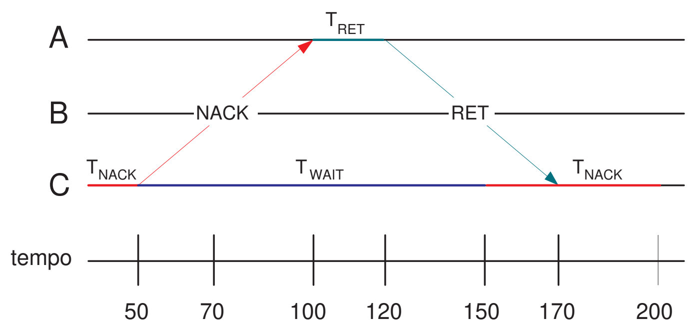

Figura 4.17: Análise da supressão do reenvio de NACK

Capítulo 5

O Modelo Multicast Confiável

ESTE capítulo apresenta a implementação do modelo que descreve a biblioteca Multicast Confiável RML.

5.1 Ambiente de Modelagem Tangram II

A análise e a modelagem de sistemas é uma tarefa essencial no processo de desenvolvimento de um sistema de computação e comunicação, onde através dos modelos pode ser avaliada a eficiência do sistema sob considerações e comparações de diferentes alternativas de desenvolvimento.

A análise do desempenho e da confiabilidade de sistemas ainda em desenvolvimento, ou até mesmo na fase de projeto, é uma das grandes vantagens do uso de modelagem. Diversos ambientes integrados e ferramentas de análise e modelagem foram desenvolvidos para dar suporte ao processo de modelagem. O modelo criado nesta dissertação foi implementado na ferramenta TANGRAM II [9, 26], desenvolvida na Universidade Federal do Rio de Janeiro (UFRJ) com a participação da Universidade da Califórnia Los Angeles (UCLA/USA) e de outras universidades. A ferramenta é distribuída gratuitamente pela Internet [1].

O TANGRAM II é um ambiente de modelagem desenvolvido com propósitos educacionais e de pesquisa, provendo flexibilidade na interface com o usuário. Apesar de possuir um paradigma genérico suficiente para suportar a descrição de sistemas gerais, o objetivo principal é fornecer suporte à descrição de sistemas de computação e comunicação. O TANGRAM II é baseado em uma estrutura orientada a objetos e possui uma sofisticada interface gráfica de domínio público denominada TGIF, *Tangram Graphic Interface Facility* [11], desenvolvida na UCLA. A versão atual está implementada em C e C++ .

Um modelo é uma abstração, que é uma visão simplificada do sistema em estudo, mas projetado com o propósito de capturar o máximo do comportamento real do sistema, suficiente para fornecer prognósticos precisos dentro de uma tolerância aceitável. Em muitos casos, modelos possuem certas vantagens óbvias sobre medidas feitas em um sistema real. Se o sistema a ser estudado ainda não existir, o modelo é a única alternativa prática para análise deste sistema.

O paradigma de modelagem adotado foi proposto em [6] e é orientado a objetos. De acordo com este paradigma, os modelos são representados por um conjunto de objetos que podem interagir entre si através do envio e do recebimento de mensagens. O comportamento dos objetos pode ser definido em eventos e mensagens associados a condições e ações. O estado interno de um objeto é armazenado por um conjunto de variáveis.

Eventos são habilitados para disparo por um objeto quando a condição para este evento disparar torna-se verdadeira. O tempo de disparo de cada evento obedece a uma distribuição de probabilidade. A ferramenta possui as seguintes distribuições: exponencial, determinística, erlangiana, gaussiana, lognormal, uniforme, pareto, FBM, FARIMA, weibull e *file* onde as amostras são lidas de um arquivo. As amostras de tempo de disparo dos eventos são geradas pela ferramenta, desde que a condição associada esteja satisfeita no tempo atual, e a execução do evento só acontecerá se esta condição permanecer válida no tempo de disparo. As condições para o disparo de eventos podem ser descritas em função do estado interno do objeto.

Mensagens são abstrações criadas para possibilitar a interação entre os objetos e são trocadas em tempo zero. Quando um evento é disparado ou uma mensagem é recebida, um conjunto de ações é executado com uma determinada probabilidade.

O tempo de execução destas ações também é zero. No final da execução de uma ação, o objeto pode ter seu estado alterado e mensagens podem ter sido enviadas a outros objetos. Estas mensagens serão tratadas pela ferramenta ainda em tempo zero.

Para criar um modelo, o usuário pode usar diferentes tipos de objetos pré-criados em uma biblioteca que acompanha a ferramenta TANGRAM II, ou criar um objeto particular. Depois de criado o objeto, o usuário precisa especificar os eventos, as ações associadas ao recebimento de mensagens, as variáveis de estado e outras especificações, que ao todo irão compor a funcionalidade do objeto. As medidas de interesse podem ser obtidas através da resolução do modelo criado. Esta resolução pode ser feita de forma matemática (analítica) ou simulada.

Note-se que para um modelo ser criado é necessário o conhecimento do sistema que está sendo modelado, assim como as facilidades e características da ferramenta de modelagem usada.

5.1.1 Simulação versus Modelos Matemáticos

Simulação é a técnica mais utilizada para modelar sistemas reais e muitas vezes esta é a única forma de avaliar sistemas grandes e complexos. Modelos matemáticos são promissores pois podem fornecer soluções fechadas, que são as formas mais desejáveis de solução, porém são conseguidas somente para modelos mais simples [15].

A qualidade dos resultados de simulação de um modelo é comumente expressa em termos de intervalos de confiança. O cálculo deste intervalo evita que se tenha que simular o sistema por um intervalo de tempo infinito para que as estatísticas possam convergir.

As maiores diferenças entre simulação e modelos matemáticos são:

- simulações são programas que, comparados com modelos matemáticos, são dispendiosos para se construir, difíceis de mostrar que estão corretos e caros em termos de ciclo de execução em CPU. Não precisa ser armazenado o espaço de estados do modelo.

- modelos matemáticos necessitam de menos tempo para sua construção e são menos suscetíveis a erros. Porém, modelos realísticos complexos possuem um espaço de estados muito grande, tornando proibida a sua solução pois ultrapassam as limitações de memória.

5.2 Criação de Modelos

Primeiramente o usuário deve definir os objetos que compõem o sistema. Após este passo, os seguintes itens devem ser definidos para que os objetos possam interagir entre si:

- as variáveis de estado;
- os eventos e as condições para que os eventos sejam habilitados;
- as ações referentes a cada um destes eventos;
- as ações associadas ao recebimento de cada mensagem pelo objeto;
- as possíveis recompensas associadas a um ou mais estados.

O usuário deverá então selecionar o módulo de modelagem *Modelling Environment* na Figura 5.1, que deverá abrir a interface de modelagem da Figura 5.2.

Cada etapa completada pelo usuário na criação e análise do modelo habilita alguma figura da interface do ambiente de modelagem tornando-a colorida. Portanto, inicialmente, todas as figuras da interface não são coloridas, como ilustra a Figura 5.2.

Para iniciar a criação do modelo, o usuário deve pressionar o botão *new* na interface de ambiente de modelagem e nomear o novo modelo a ser criado. Após a conclusão desta etapa, a figura que representa o TGIF estará colorida, ou seja,

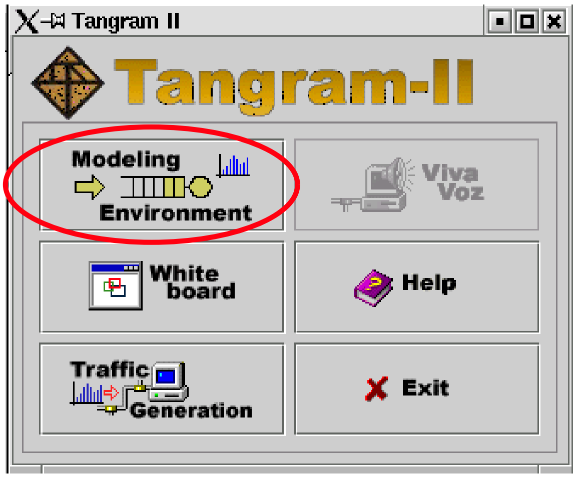

Figura 5.1: Interface principal da ferramenta TANGRAM II

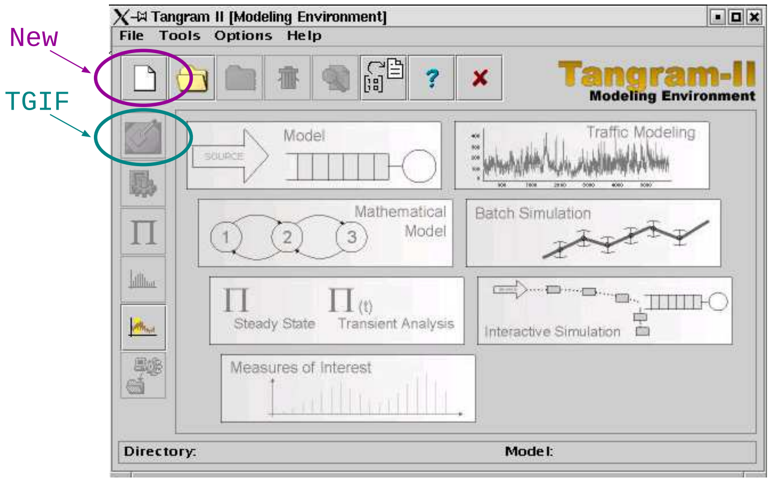

Figura 5.2: Interface do ambiente de modelagem

habilitada para uso. O modelo deve ser especificado no TGIF, cuja interface é ilustrada na Figura 5.3.

A definição de um objeto consiste na descrição de seis atributos: declaração, inicialização, eventos, variáveis de estado, mensagens e recompensas.

Declaração

Este atributo é responsável pela declaração das variáveis, constantes e parâmetros usados no objeto. Onde os seguintes tipos podem ser usados:

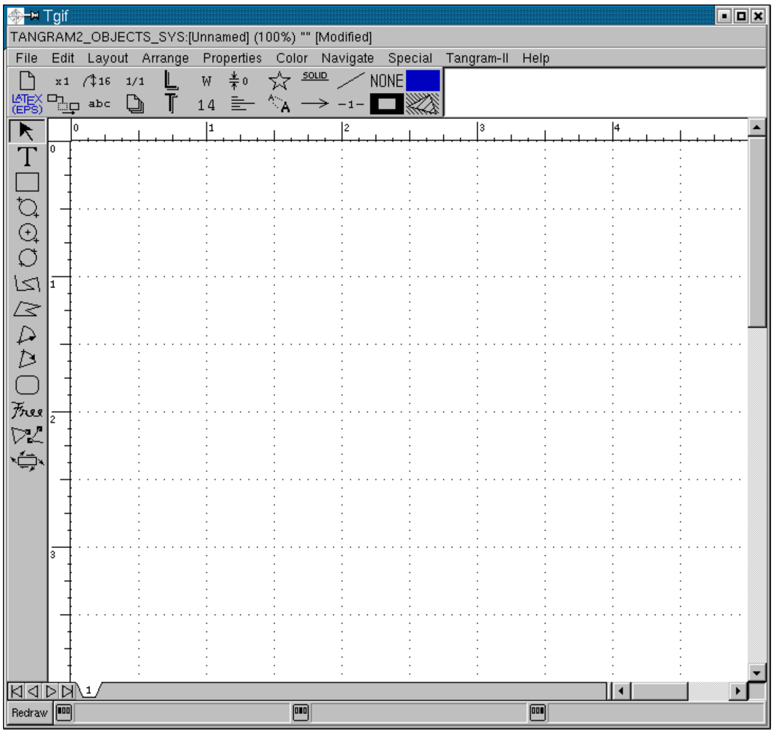

Figura 5.3: Interface TGIF (TANGRAM *Graphic Interface Facility*)

State: declara variáveis que representam o estado interno dos objetos.

Integer, *Float*: especifica constantes numéricas ou parâmetros.

Object: referencia outros objetos do modelo.

Port: define portas usadas na comunicação entre os objetos.

Inicialização

Este atributo é responsável pela inicialização das variáveis de estado e pela definição das constantes.

Eventos

Este atributo é responsável pela especificação de todos os eventos do objeto. Um evento pode ser definido através de um nome, um tipo de distribuição e seus parâmetros, e as ações por ele definidas somente serão executadas se a condição para tal for satisfeita. Segundo as regras da ferramenta, o valor de uma variável de estado pode ser alterada, através da função *set_st*, somente no final das ações.

A clonagem de eventos é feita quando se deseja gerar mais de uma amostra de um evento. Um evento é clonado através da função *clone_ev*. Para que as ações, especificadas no evento clonado, sejam executadas é necessário que a condição do evento seja satisfeita no momento que ele é clonado e quando ele for executado.

Variáveis de Estado

Este atributo é responsável pela listagem das variáveis de estado que deverão ser mostradas durante a simulação interativa. Cada passo da simulação atualiza os valores das variáveis de estado listadas neste atributo. O usuário pode alterar o valor de alguma variável de estado, forçando uma mudança no caminho amostral.

Mensagens

Este atributo é responsável pelo tratamento de todas as mensagens que podem ser recebidas pelo objeto. Quando uma mensagem específica é recebida, a ação associada a esta mensagem é executada. As mensagens são identificadas pela constante *Port*, que indica a porta que uma mensagem está sendo recebida.

Recompensas

Este atributo é responsável pela especificação de recompensas de taxa e de impulso. Diversas medidas de interesse podem ser obtidas através do uso das recompensas, como: perdas, tempo de recuperação de um pacote perdido, número de falhas, tempo de reparo, vazão, utilização, tempo operacional e muitas outras medidas. Um objeto pode ter várias recompensas definidas, possibilitando a obtenção de diversas medidas simultaneamente.

Se uma recompensa de taxa r_i for associada a um estado i, então o sistema ganha a recompensa r_i por unidade de tempo gasto no estado i. A recompensa de impulso é associada a transições. Se uma recompensa r_{ij} é associada à transição do estado i para o estado j, então o sistema ganha a recompensa r_{ij} toda vez que o sistema fizer uma transição de i para j.

No TANGRAM II, as recompensas de taxa estão associadas, através de condições, aos estados dos objetos. Após a definição do nome da recompensa, a condição e o valor devem ser definidos, onde a primeira define o conjunto de estados em que a recompensa irá acumular o valor indicado.

A utilização de um objeto que simula uma fila, por exemplo, pode ser obtida através da definição de uma recompensa de taxa que acumule o valor unitário sempre que haja pacotes a serem servidos. A sintaxe da definição desta recompensa de taxa seria:

```
Rewards=
   rate_reward = utilizacao_fila
      condition = (fila > 0)
      value = 1;
```

As recompensas de impulso são definidas através de um nome, da especificação do evento ou da mensagem que a recompensa estará associada e de uma ou mais ações que, ao serem executadas, farão com que a recompensa acumule o valor determinado.

O número de pacotes transmitidos por uma fonte pode ser obtido através de uma recompensa de impulso associada ao evento que gera os pacotes, por exemplo o evento *geração_pacote*. A sintaxe da definição desta recompensa de impulso seria:

```
Rewards=
   impulse_reward = num_pacotes
      event = geracao_pacote,1
      value = 1;
```

Cada evento ou execução de mensagens pode estar associado a uma ou mais ações. No exemplo da sintaxe acima, a recompensa se refere à primeira ação dentro do evento *geração_pacote*. O número ao lado do nome do evento indica a ação à qual a recompensa se refere, e é relativo à ordem de aparição das ações no código do modelo, iniciando com 1.

5.3 Implementação do Modelo Multicast Confiável

O modelo foi implementado com o objetivo de facilitar a análise de desempenho da biblioteca Multicast confiável RML. Devido à grande quantidade de variáveis de estado e a complexidade do modelo, as análises foram feitas através de simulações.

As simulações despendem um tempo significativamente menor quando comparadas com os testes reais. Para obter medidas do comportamento da RML em uma apresentação com a duração de 1h no TGWB, por exemplo, seria necessário que alguns usuários estivessem em locais distintos da rede utilizando a ferramenta. Além do tempo gasto para obtenção das medidas, recursos da rede seriam alocados para realização dos testes. Para rodar o modelo, simulando uma apresentação com a duração de 1h, é gasto em média 6min em uma máquina Pentium III ou IV (ou processadores equivalentes) com mais de 128Mb de memória. Além disso, é possível estudar cenários não disponíveis na realização dos testes.

Inúmeras topologias poderiam ser descritas através do modelo, porém foi necessária a escolha de uma topologia para a realização das simulações. Como as seguintes universidades usadas para testes foram a UFRJ (Universidade Federal do Rio de Janeiro), a UFF (Universidade Federal Fluminense), a UFJF (Universidade Federal de Juiz de Fora) e a UMASS (Universidade de Massachusetts), o modelo descreve o ambiente de comunicação Multicast entre estas universidades.

Foram implementados dois modelos que descrevem a transmissão Multicast confiável, porém, com a disposição dos objetos um pouco diferente. O modelo com o *spliter* (Figura 5.4) representa a transmissão Multicast entre todos os membros do grupo, onde o *spliter* seria o último roteador comum entre o emissor e os receptores; e o modelo com os *proxies* (Figura 5.5) simula os testes que serão apresentados no Capítulo 6, onde a transmissão entre os *proxies* é feita via conexão Unicast (maiores detalhes sobre os *proxies* encontram-se no Capítulo 6). O funcionamento dos objetos em comum são idênticos, porém cada modelo tem sua especialidade, o modelo com o *spliter* é mais genérico e simula o suporte dos roteadores da Internet para a transmissão Multicast, no entanto o modelo com os *proxies* descreve o ambiente de testes utilizado para análise do comportamento da biblioteca durante uma apresentação na ferramenta TGWB. Por terem as mesmas funcionalidades os objetos que representam as universidades e os canais são os mesmos, somente o *spliter* e o *proxy* possuem o funcionamento um pouco diferente. Os detalhes sobre os objetos implementados nos modelos serão explicados na Seção 5.3.1.

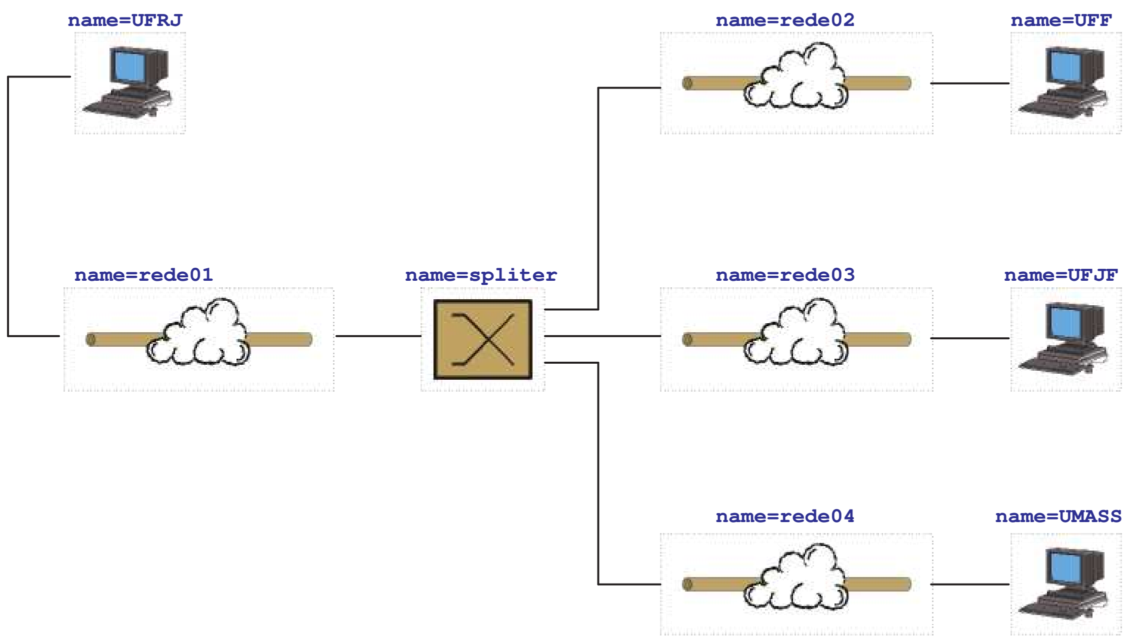

Figura 5.4: O modelo RML com o uso do *spliter*

As simulações consideram apenas um emissor de dados: o objeto UFRJ, e três receptores: UFF, UFJF e UMASS. O fato de simular os modelos com apenas um emissor não altera a análise das medidas de interesse apresentadas na Seção 5.3.5.

Como as mensagens de controle são pacotes pequenos quando comparados aos pacotes de dados e as retransmissões, a probabilidade de perda é menor. Sendo assim, no modelo da RML, para simplificação, é considerado que as mensagens de controle NACKs e *refreshs* nunca serão perdidas. Também com o objetivo de simplificar, o modelo só considera a ocorrência de perdas e atrasos nos objetos que simulam os canais e os atrasos de ida e volta entre duas universidades são considerados iguais. Entretanto estas simplificações podem facilmente ser descartadas.

A unidade de tempo considerada para os modelos é milisegundos (*ms*). A uti-

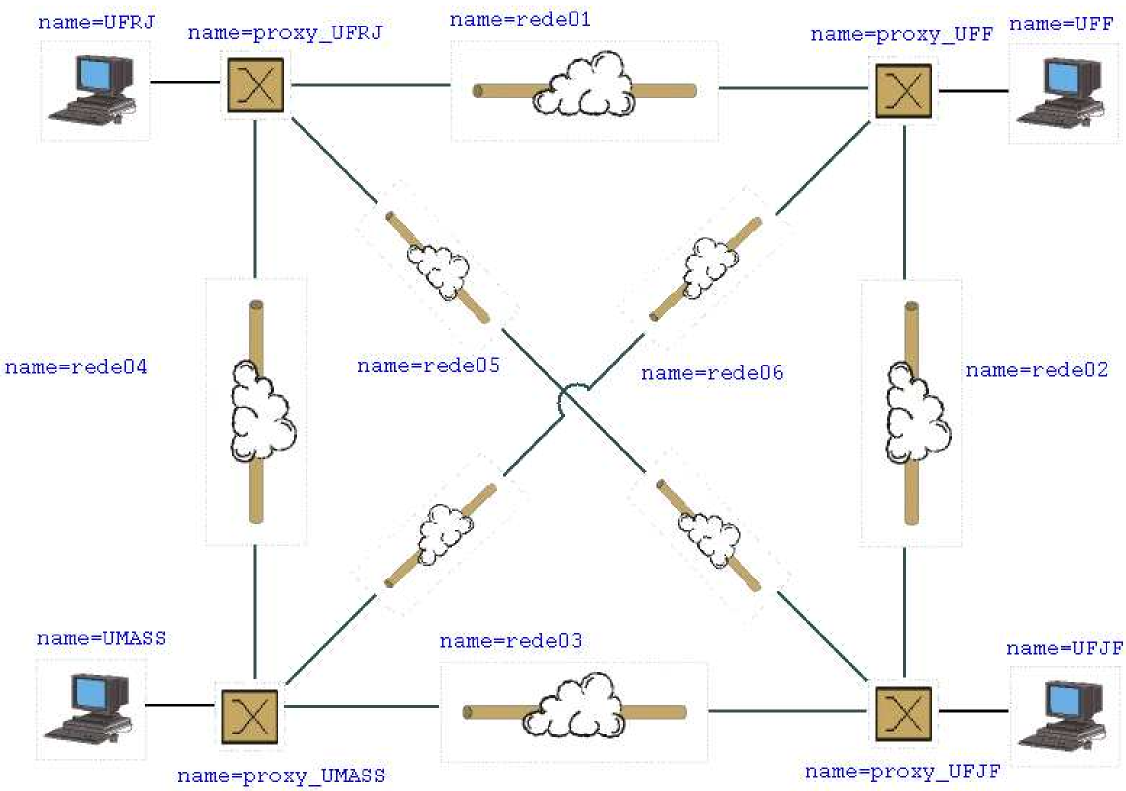

Figura 5.5: O modelo RML com uso dos *Proxies*

lização da distribuição uniforme, em alguns eventos, baseia-se na implementação proposta pelo protocolo SRM [19]. O código dos modelos encontra-se no Apêndice A.

5.3.1 Objetos

Foram criados catorze objetos no modelo com o *proxy* e nove no modelo com o *spliter*, são eles:

- quatro universidades em ambos os modelos;
- um *spliter* e quatro canais de comunicação no modelo com o *spliter*;
- quatro *proxies* e seis canais de comunicação no modelo com o *proxy*.

Os objetos que representam as universidades têm a função de enviar e receber dados Multicast, igualmente como foi implementado na biblioteca RML. Estes objetos interagem entre si através do envio de dados, retransmissões, NACKs e *refreshs*.

Os objetos que representam os canais de comunicação simulam o atraso e as perdas na rede. Quando uma mensagem é recebida pelo objeto através de uma porta, com a probabilidade (1 - *prob_perda*) esta mensagem é colocada em uma fila de espera antes de ser enviada por um evento para a outra porta do objeto, simulando o atraso no canal; ou com a probabilidade *prob_perda* o objeto não faz nada, simulando a perda no canal.

O objeto *spliter* simplesmente repassa a mensagem para todas as suas portas, exceto para a porta pela qual a mensagem foi recebida.

Os objetos *proxies* quando recebem uma mensagem vinda por uma porta conectada com o canal, a repassam para a universidade e quando recebe uma mensagem vinda da porta que o conecta a universidade, a repassa para cada umas das portas que irão levá-la aos canais.

O comportamento de cada um destes objetos pode ser definido através dos eventos e das mensagens associados a condições e ações, conforme explicitado nas Seções 5.3.3 e 5.3.4.

5.3.2 Estruturas de Dados

As mensagens enviadas pelo modelo possuem as estruturas representadas pela Figura 5.6, onde o primeiro campo de cada mensagem é o seu tipo e o segundo é a identificação do emissor da mensagem. Se o tipo da mensagem for DADO ou RET (retransmissão), então o terceiro campo conterá o valor do *sn* enviado. Se o tipo da mensagem for NACK, então o terceiro campo em seguida conterá o valor do *sn* do pacote que requisita retransmissão. Se o tipo da mensagem for REFRESH, então o terceiro campo conterá o valor do último *sn* enviado.

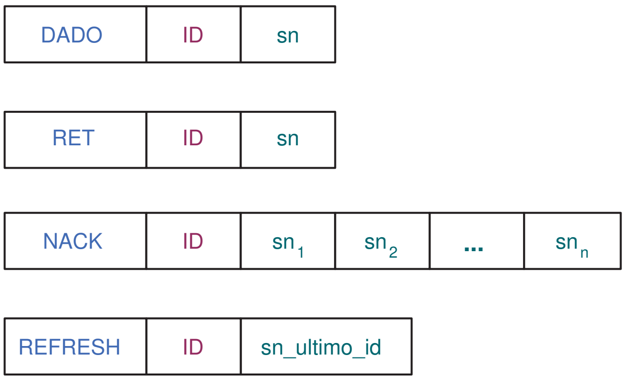

Figura 5.6: Tipos de mensagens enviadas pelo modelo RML

A *cache* é uma estrutura importante no modelo, similar a *cache* implementada na biblioteca RML, tem a função de guardar os dados recebidos, as informações sobre quais pacotes devem ser enviados os NACKs e quais estão esperando por retransmissão. As seguintes variáveis auxiliam o gerenciamento de acesso à *cache* :

$sn\_inicial$: indica qual o valor do sn da posição inicial da *cache*.

$pos\_inicial$: indica a posição inicial da *cache*.

pos_ultimo_id : indica a posição do último sn identificado.

pos_ultimo_seq : indica a posição do último sn em sequência.

O cálculo para saber qual o sn do dado de uma determinada posição e qual a posição de um determinado dado são feitos segundo as seguintes equações:

$$sn = sn\_inicial + posicao \tag{5.1}$$

$$posicao = sn \bmod tamanho\_da\_cache \tag{5.2}$$

A *cache* armazena as informações indicadas na Tabela 5.1.

Informação	**Significado**
DADO	indica o recebimento do dado
NACK	indica que deve ser enviado um NACK
RET_WAIT	indica a espera pela retransmissão do dado

Tabela 5.1: Informações armazenadas na *cache*

A Figura 5.7 é uma representação simplificada da *cache* implementada no modelo, com apenas 5 posições. Vamos supor que um objeto receba os pacotes com os *sns* de 0 a 6 e depois receba o pacote com o $sn = 9$. Primeiramente, ao receber os pacotes com os *sns* de 0 a 4, o objeto mantém o valor do $sn\_inicial = 0$, indicando que o menor pacote armazenado na *cache* é o pacote com o $sn = 0$. Porém, como a *cache* possui apenas 5 posições, com o recebimento do pacote com o $sn = 5$, este substitui o pacote de $sn = 0$ e o valor da variável $sn\_inicial$ passa a valer 1, pois o pacote com o $sn = 1$ torna-se o menor pacote armazenado na *cache* até o momento. Similarmente, o pacote com o $sn = 6$ ao ser recebido, substitui o pacote de $sn = 1$ e o valor da variável $sn\_inicial$ passa a valer 2. O recebimento do pacote com o $sn = 9$ é indicado na posição 4 da *cache* através da palavra "DADO". A posição deve ser 4 pois, segundo a equação 5.2, $9 \ mod \ 5 = 4$. O valor da variável $sn\_inicial$, após o recebimento deste pacote, é 5, pois o pacote com o $sn = 5$ torna-se o menor pacote armazenado na *cache*.

O recebimento fora de sequência do pacote com $sn = 9$ faz com que o objeto detecte a perda dos pacotes com os *sns* 7 e 8. Como mostra a Figura 5.7, quando o objeto detecta a perda de pacotes, esta é indicada na *cache* através da palavra "NACK" na posição do *sn* que está esperando para enviar o NACK.

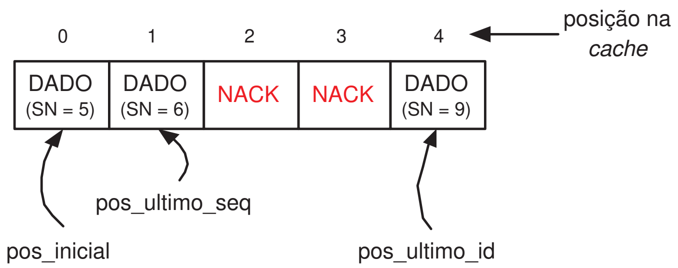

Figura 5.7: A *cache* do modelo RML

Devido às limitações do tamanho da *cache*, imposta pela ferramenta, onde um vetor pode ter no máximo 255 posições, foi implementado o vetor de retransmissões *ret*, com o objetivo de guardar os *sns* dos pacotes que deverão ser retransmitidos. Portanto, apenas com a *cache*, no modelo, não seria viável simular a *cache* da biblioteca RML que possui o valor *default* igual a 4000, que é o tamanho suficiente para comportar o recebimento dos pacotes provenientes da captura de uma tela pelo usuário. Então, o vetor *ret* simula a *cache* de 4000 posições da biblioteca. Uma vez que são enviados e recebidos apenas o valor do *sn* do pacote, fica fácil retransmitir algum pacote perdido, bastando saber se este pacote ainda estaria na *cache* se esta tivesse 4000 posições e guardar valor do *sn* requisitado no vetor *ret* para futura retransmissão. Obviamente, o vetor *ret* também possui apenas 255 posições, porém os *sns* não são mantidos em sequência como na *cache*, podendo armazenar até 255 pacotes fora de sequência para serem retransmitidos.

No modelo foram criadas recompensas para a análise do comportamento do vetor *ret*, onde foi possível variar o tamanho deste vetor e analisar se os pedidos de retransmissão recebidos eram escalonados no vetor, para futura retransmissão, ou se o vetor estava cheio de pacotes para retransmitir e, consequentemente, não escalonou a retransmissão. Os resultados se encontram nas Tabelas 5.2, 5.3, 5.4 e 5.5, onde o comportamento do vetor de retransmissões, *ret*, foi analizado de acordo com as variações na perda e no tamanho do vetor em, respectivamente, 10, 20, 50 e 255 posições. Para cada tamanho de vetor foram feitas duas simulações. A primeira, identificada no lado esquerdo da tabela, simula as perdas obtidas a partir dos testes realizados com a biblioteca RML e a segunda, identificada no lado direito da tabela, possui valores maiores em relação ao valor das perdas obtidas nos testes, possibilitando a análise da sensibilidade do escalonamento das retransmissões no vetor *ret* diante do aumento das perdas. As simulações foram feitas no modelo com o objeto *spliter*, porém devido ao igual funcionamento do vetor *ret* em ambos os modelos, pode-se considerar os resultados válidos para o modelo com o *proxy* também. Os atrasos considerados nestas simulações foram obtidos através de RTTs coletados durante um dia, sendo eles: 500ms no canal entre o objeto *spliter* e o UFF, 127ms no canal entre o objeto *spliter* e o UFJF e, 125ms no canal entre o objeto *spliter* e o UMASS. Pode ser observado que o atraso em relação a UFF foi maior do que o atraso em relação a UMASS, isso ocorreu devido à problemas, não identificados, no recebimento de dados na UFF. As perdas consideradas encontram-se entre o objeto *spliter* e as universidades receptoras.

Vetor Retransmissão com 10 posições								
objeto	UFRJ	UFF	UFJF	UMASS	UFRJ	UFF	UFJF	UMASS
perda	0%	13%	26%	12%	0%	40%	60%	30%
prob. de escalonar a retrans.	0.54	0.27	0.6	0.57	0.31	0.1	0.49	0.34

Tabela 5.2: Resultados da análise do comportamento do vetor de retransmissões com **10** posições

Vetor Retransmissão com 20 posições								
objeto	UFRJ	UFF	UFJF	UMASS	UFRJ	UFF	UFJF	UMASS
perda	0%	13%	26%	12%	0%	40%	60%	30%
prob. de escalonar a retrans.	0.72	0.46	0.86	0.72	0.5	0.22	0.64	0.54

Tabela 5.3: Resultados da análise do comportamento do vetor de retransmissões com **20** posições

Vetor Retransmissão com 50 posições								
objeto	UFRJ	UFF	UFJF	UMASS	UFRJ	UFF	UFJF	UMASS
perda	0%	13%	26%	12%	0%	40%	60%	30%
prob. de escalonar a retrans.	0.97	0.87	1	0.94	0.76	0.52	0.97	0.77

Tabela 5.4: Resultados da análise do comportamento do vetor de retransmissões com **50** posições

Vetor Retransmissão com 255 posições								
objeto	UFRJ	UFF	UFJF	UMASS	UFRJ	UFF	UFJF	UMASS
perda	0%	13%	26%	12%	0%	40%	60%	30%
prob. de escalonar a retrans.	1	1	1	1	1	1	1	1

Tabela 5.5: Resultados da análise do comportamento do vetor de retransmissões com **255** posições

Podemos observar que o objeto UFJF manteve uma maior probabilidade de escalonar os pedidos de retransmissão. Isso deve-se ao fato de possuir a maior perda, tornando-o o objeto que mais enviou NACKs, o que, consequentemente, provocou uma maior quantidade de pedidos de retransmissão nos demais objetos. Também pode ser observado que o objeto UFF manteve a menor probabilidade de sucesso no escalonamento das retransmissões, pois uma vez que os temporizadores estão baseados nos atrasos da rede e o objeto em questão possui o maior atraso, o intervalo de tempo para retransmitir um pacote neste objeto é maior do que nos demais. Assim o objeto UFF pode demorar mais que os outros para retransmitir, mantendo por mais tempo um pedido de retransmissão escalonado no vetor, o que aumenta a chance de não poder escalonar um novo pedido por falta de espaço, momentâneo, no vetor. Com o aumento da perda na segunda simulação, o número de NACKs enviados pelos objetos também é aumentado, logo aumenta a probabilidade de um pedido de retransmissão não poder ser escalonado no vetor, ou seja, diminui a probabilidade de sucesso no escalonamento de um pedido de retransmissão. Os resultados apresentados nas Tabelas 5.2, 5.3, 5.4 e 5.5 evidenciam que um vetor com

255 posições pode escalonar os pedidos de retransmissão 100% das vezes em que é requisitado.

A Figura 5.8 é um exemplo do vetor de retransmissões *ret*, onde:

pos_ret_inicial: indica a posição do próximo *sn* a ser retransmitido.

pos_ret_final: indica a posição do último *sn* escalonado para ser retransmitido.

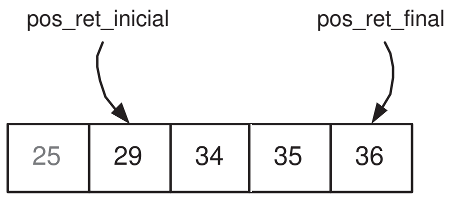

Figura 5.8: O vetor de retransmissões do modelo RML

Após a retransmissão de um pacote, a variável *pos_ret_inicial* é incrementada. Caso uma nova retransmissão seja escalonada, a variável *pos_ret_final* será incrementada e o *sn* a ser retransmitido deverá ser incluído na posição *pos_ret_final*. No exemplo da Figura 5.8, o pacote com o $sn = 25$ já foi retransmitido e o próximo *sn* a ser retransmitido será o 29.

As estruturas *fila_msg*, *Fila01* e *Fila02* armazenam temporariamente as mensagens recebidas pelo objeto, possibilitando o funcionamento do evento Tratar_Msg e simulando o atraso no canal, respectivamente.

5.3.3 Eventos

Os seguintes eventos foram implementados no modelo:

Enviar_Dados

Evento com função de enviar dados Multicast para o grupo. O evento não está habilitado em todos os objetos, somente no objeto que simula o emissor.

O evento possui distribuição uniforme e o intervalo de tempo para execução do evento encontra-se entre 30 e 60 s, simulando o tempo que um usuário levaria para trocar cada página de uma apresentação no TGWB.

O número de pacotes enviados pode variar cada vez que o evento é executado. Como o objetivo é simular uma apresentação no TGWB, foi analisada a quantidade de pacotes que o TGWB manda em uma apresentação, considerando que 70% das apresentações são simples e sem figuras, 25% são com poucas figuras e 5% são com muitas figuras, e sabendo que quanto mais figuras maior é a quantidade de pacotes a serem transferidos, segue o resultado da análise do envio de dados:

- em 70% das vezes são enviados em média 25 pacotes;
- em 25% das vezes são enviados em média 100 pacotes;
- em 5% das vezes são enviados em média 200 pacotes.

Portanto, o evento Enviar_Dados envia 25 pacotes com a probabilidade 0,7, 100 pacotes com a probabilidade 0,25 e 200 pacotes com a probabilidade 0,05.

A condição para o evento é sempre verdadeira para o objeto emissor, ou seja, o evento está sempre habilitado, devendo ocorrer segundo sua distribuição e intervalo de tempo. E a condição é sempre falsa para os objetos receptores, não estando jamais habilitados para enviar dados.

Tratar_Msg

Toda vez que uma mensagem é recebida pelos objetos UFRJ, UFF, UFJF e UMASS, a variável *msg_cont* é incrementada, indicando que existe uma mensagem a ser processada pelo objeto. A condição para execução do evento Tratar_Msg é ter mensagem para tratar, ou seja, ter a variável $msg\_cont > 0$. Este evento se encontra apenas nos objetos UFRJ, UFF, UFJF e UMASS.

O evento Tratar_Msg retira a mensagem da fila *fila_msg*, verifica qual o seu tipo e qual ação deve ser tomada dependendo do tipo. Assim como na RML, quando um

NACK é recebido, o objeto deve verificar em sua *cache* se deseja enviar um NACK para os mesmos pacotes requisitados. Caso positivo, então, o objeto tem o envio do NACK suprimido e deve indicar na *cache* que está esperando pela retransmissão. Se na *cache* estiver indicando que o objeto já enviou o NACK para os mesmos pacotes e está esperando por retransmissões, então nada deverá ser feito. Porém, se nenhum dos casos anteriores forem satisfeitos, o objeto deverá verificar a existência do pacote requisitado na *cache* e escalonar o evento para sua retransmissão. Quando o objeto recebe um dado, deve verificar se está fora de sequência; caso esteja, deve então ser escalonado o evento para envio do NACK requisitando os pacotes perdidos. Quando um *refresh* é recebido, é feita a mesma verificação utilizada para o recebimento de um dado. Enfim, todas as ações tomadas no recebimento das mensagens pela biblioteca RML são simuladas no modelo.

Quando uma mensagem é recebida, as variáveis auxiliares $aux\_T\_RET\_MIN$ e $aux\_T\_RET\_MAX$ recebem o valor do limite mínimo e máximo, respectivamente, do intervalo do temporizador referente ao emissor do NACK. Porém, se o evento Enviar_Retransmissão for clonado neste instante, as variáveis T_RET_MIN e T_RET_MAX, que determinam os limites mínimo e máximo do intervalo do temporizador deste evento, ainda não foram atualizadas. Isto acontece porque todas as variáveis de estado somente são atualizadas no final das ações. Portanto, foi necessária a criação do evento auxiliar Tratar_Msg como será explicado a seguir.

Supondo o valor inicial das variáveis $T\_RET\_MIN = 4$ e $T\_RET\_MAX = 8$ e que o objeto UFRJ receba uma mensagem NACK do objeto UFF; supondo ainda a seguinte descrição simplificada do código do recebimento de mensagens sem o uso do evento Tratar_Msg:

```
aux_msg_cont++;

 if (aux_fila_msg[0] == NACK){
  switch (aux_fila_msg[1]){
      case UFF:
       aux_T_RET_MIN = 50;
       aux_T_RET_MAX = 100;
       break;

      case UFJF:
       aux_T_RET_MIN = 10;
       aux_T_RET_MAX = 20;
       break;

      case UMASS:
       aux_T_RET_MIN = 14;
       aux_T_RET_MAX = 28;
       break;
  }
 }

clone_ev("Enviar_Retransmissão");

set_st( "T_RET_MIN", aux_T_RET_MIN);
set_st( "T_RET_MAX", aux_T_RET_MAX);
set_st( "msg_cont", aux_msg_cont );
```

O evento Enviar_Retransmissão vai ser escalonado no intervalo entre T_RET_MIN e T_RET_MAX que são iguais a 4 e 8, respectivamente, no momento de disparo deste evento. Porém, com o uso do evento Tratar_Msg, como sugerido no código simplificado a seguir, é assegurada a atualização das variáveis T_RET_MIN e T_RET_MAX no momento de disparo do evento Enviar_Retransmissão, conforme desejado. O evento Tratar_Msg é executado segundo a taxa $tratar\_msg\_rate$ e a condição $msg\_cont > 0$ e não depende do valor das variáveis T_RET_MIN e T_RET_MAX para ser escalonado. Portanto, o valor desatualizado destas variáveis, no momento de disparo do evento, não influenciam na sua execução, pois não indicam a taxa de execução do evento. Este evento criará um clone do evento Enviar_Retransmissão, neste momento as variáveis T_RET_MIN e T_RET_MAX já estarão atualizadas com os valores 50 e 100. Segue a descrição simplificada do código do recebimento de mensagens com o uso do evento Tratar_Msg:

```
aux_msg_cont++;

 if (aux_fila_msg[0] == NACK){
  switch (aux_fila_msg[1]){
      case UFF:
       aux_T_RET_MIN = 50;
       aux_T_RET_MAX = 100;
       break;

      case UFJF:
       aux_T_RET_MIN = 10;
       aux_T_RET_MAX = 20;
       break;

      case UMASS:
       aux_T_RET_MIN = 14;
       aux_T_RET_MAX = 28;
       break;
  }
 }

clone_ev("Tratar_Msg");

set_st( "T_RET_MIN", aux_T_RET_MIN);
set_st( "T_RET_MAX", aux_T_RET_MAX);
set_st( "msg_cont", aux_msg_cont );
```

A Figura 5.9 demonstra o intervalo do temporizador para disparo do evento Enviar_Retransmissão nas duas situações descritas anteriormente, sem e com o uso do evento Tratar_Msg, respectivamente.

Enviar_Nack

O evento Enviar_Nack só está habilitado nos objetos UFF, UFJF e UMASS. Sua função é verificar na *cache* quais as posições que contém o valor 2, pois indicam que o *sn* da posição foi perdido e que deverá ser enviado um NACK. O valor 2 é substituído pelo valor 5, que sinaliza a espera pela retransmissão requisitada, e o NACK é enviado contendo todos os *sns* para os quais deseja-se receber retransmissões. Cada

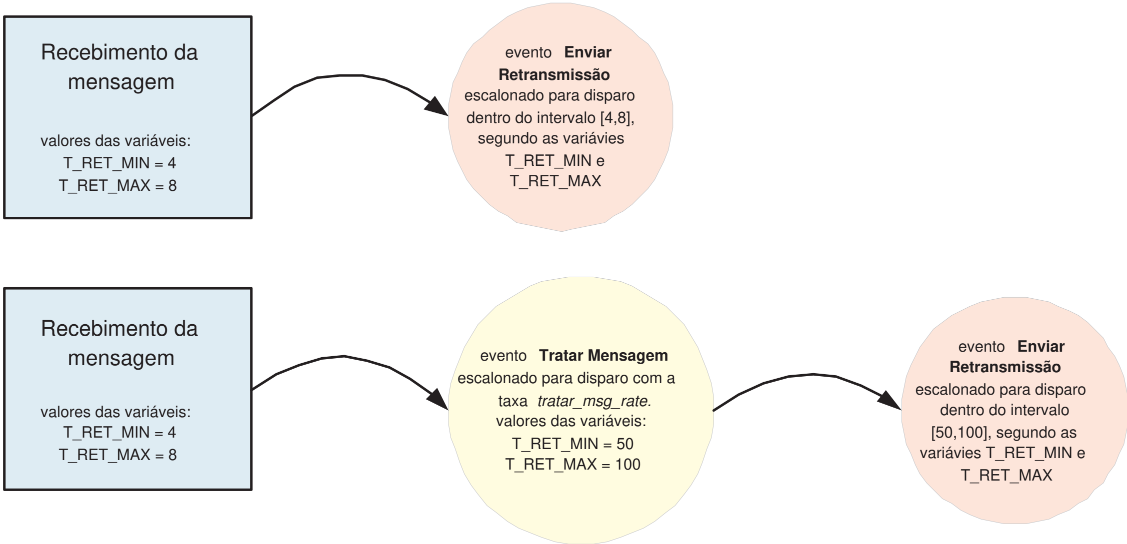

Figura 5.9: Intervalo do temporizador para o evento Enviar_Retransmissão

NACK enviado pode conter até 64 *sns*, indicando 64 pedidos de retransmissões, assim como foi implementado na biblioteca RML.

Toda vez que for detectada uma perda, a variável *nack_cont* é incrementada, indicando que existe mais um NACK a ser enviado pelo objeto. E a condição para execução do evento Enviar_Nack é ter NACK para enviar, ou seja, ter a variável $nack\_cont > 0$. Cada *sn* colocado na mensagem NACK a ser enviada, decrementa a variável *nack_cont* e incrementa a variável *ret_wait_cont*, que habilitará o evento Espera_Retransmissão.

O evento possui distribuição uniforme, variando os limites inferior e superior do intervalo, segundo a definição da biblioteca RML apresentada no Capítulo 4:

$$T_{NACK} = \{\mathrm{A} * delay_{receptor,emissor},\ (\mathrm{A} + \mathrm{B}) * delay_{receptor,emissor}\}$$

onde $delay_{receptor,emissor}$ é o atraso entre o membro receptor e o membro emissor do dado transmitido e A e B são constantes.

Os valores das variáveis A e B podem variar, tornando flexível para que o usuário obtenha medidas de acordo com as variações destas variáveis.

Se uma retransmissão for recebida no momento em que o objeto está esperando para enviar um NACK correspondente, então a variável *nack_cont* é decrementada e a posição deste pacote na *cache* recebe o dado retransmitido, com o isso o NACK que estava escalonado não é enviado, simulando sua supressão.

Espera_Retransmissão

O evento Espera_Retransmissão também só está habilitado nos objetos UFF, UFJF e UMASS. Sua função é verificar na *cache* quais as posições que contém o valor 5, pois indicam que o *sn* da posição está esperando por uma retransmissão. O valor 5 é substituído pelo valor 2, que sinaliza que a retransmissão requisitada não foi recebida e que deve ser enviado um novo NACK para este *sn*.

O evento estará habilitado se a variável $ret\_wait\_cont > 0$. A variável *ret_wait_cont* deve ser decrementada e a variável *nack_cont* incrementada, a cada mudança do valor 5 para 2 em alguma posição da *cache*.

O evento possui distribuição uniforme, variando os limites inferior e superior do intervalo, segundo a definição da biblioteca RML apresentada no Capítulo 4:

$$T_{WAIT} = \{\mathrm{C} * delay_{receptor,emissor}), (\mathrm{C} + \mathrm{D}) * delay_{receptor,emissor}\}$$

onde $delay_{receptor,emissor}$ é o atraso entre o membro receptor e o membro emissor do dado transmitido e C e D são constantes.

Para tentar evitar o reenvio prematuro de NACKs, o tempo que se espera para enviar um novo NACK deve ser maior que a soma de quatro fatores: o tempo do NACK anterior chegar em algum membro, o tempo de processamento deste NACK, o tempo de espera para envio da retransmissão e o tempo desta retransmissão chegar no membro que enviou o NACK. Com isso, tenta-se garantir que o reenvio do NACK só seja feito no caso da perda do NACK anteriormente enviado ou da retransmissão referente a este NACK.

A Figura 5.10 ilustra o cálculo abaixo, que para simplificação, considerou que o tempo de processamento do NACK é nulo e que os atrasos entre o receptor e o emissor são iguais:

$$\min\{T_{WAIT} + T_{NACK}\} > d_1 + T_{PROC} + \max\{T_{RET}\} + d_2$$

Considerando $d_1 = d_2 =$ d e $T_{PROC} \approx 0$, temos:

$$(C * d) + (A * d) > 2d + (E + F)\, d$$
$$(C + A) > 2 + (E + F)$$
$$C > 2 + E + F - A$$

onde

T_{WAIT} : é o intervalo de espera por retransmissão;

T_{NACK} : é o intervalo de espera para envio do NACK;

T_{RET} : é o intervalo de espera para envio da retransmissão;

T_{PROC} : é o tempo de processamento do NACK;

d_1 : é o atraso entre o receptor e o emissor;

d_2 : é o atraso entre o emissor e o receptor.

Conclui-se que o valor do parâmetro C deve ser sempre maior que (2 + E + F - A) para tentar evitar o envio prematuro dos NACKs. Porém, o envio prematuro dos NACKs não pode ser evitado em todos os casos, pois a instabilidade da rede pode causar um aumento significativo no atraso entre os membros, fazendo com que a retransmissão demore mais tempo que o previsto para chegar e com isso que o membro que enviou o NACK acabe reenviando outro desnecessariamente.

Enviar_Retransmissão

O evento Enviar_Retransmissão está habilitado nos objetos UFRJ, UFF, UFJF e UMASS toda vez que uma mensagem NACK é recebida. Após o recebimento do

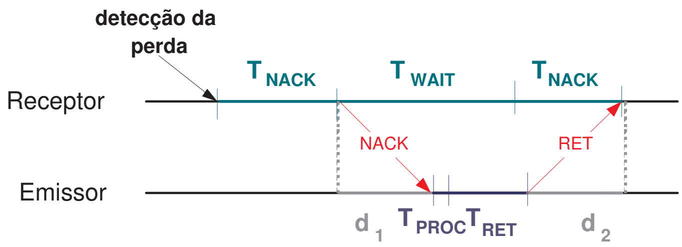

Figura 5.10: Estudo do parâmetro C para supressão do reenvio do NACK

NACK, o objeto incrementa a variável *ret_cont*. Esta variável é a condição para execução do evento, ou seja, o evento será executado se *ret_cont* > 0.

Quando este evento é executado, o *sn* da posição *pos_ret_inicial* do vetor *ret* é enviado e a variável *pos_ret_inicial* é incrementada.

O evento possui distribuição uniforme, variando os limites inferior e superior do intervalo, segundo a definição da biblioteca RML apresentada no Capítulo 4:

$$T_{RET} = \{\mathrm{E} * delay_{receptor,emissor}, (\mathrm{E} + \mathrm{F}) * delay_{receptor,emissor}\}$$

onde $delay_{receptor,emissor}$ é o atraso entre o membro receptor e o membro emissor do dado transmitido e E e F são constantes.

Se uma retransmissão for recebida no momento em que o objeto está esperando para enviar a retransmissão correspondente, então a posição deste pacote no vetor *ret* recebe o valor -1. Quando o evento Enviar_Retransmissão for executado e o *sn* do pacote a ser retransmitido for igual a -1, nenhuma mensagem deverá ser enviada, simulando a supressão de retransmissão.

Enviar_Refresh

O evento Enviar_Refresh tem a função de enviar *refreshs*, que são mensagens indicando o *sn* do último pacote enviado. O evento não está habilitado em todos os objetos, somente no objeto que simula o emissor.

O evento é determinístico e envia uma mensagem *refresh* a cada 10 segundos.

Delay01 e Delay02

Ambos os eventos têm o objetivo de simular o atraso da rede e encontram-se nos objetos que representam os canais de comunicação. São eles: rede01, rede02, rede03, rede04, rede05 e rede06.

Toda mensagem recebida com sucesso, por uma porta, pelo objeto, é armazenada temporariamente em uma fila de espera, tipo servidor infinito. Os eventos Delay01 e Delay02 têm a função de retirar a mensagem recebida desta fila e enviá-la pela outra porta.

Estes dois eventos são definidos porque os objetos que representam os canais possuem duas portas. No modelo com o *spliter*, uma das portas do canal o conecta ao emissor ou receptor e a outra ao *spliter*. Portanto, o Delay01 recebe as mensagens vindas do objeto emissor ou receptor e as envia para o objeto *spliter*, e o Delay02 recebe as mensagens vindas do objeto *spliter* e as envia para o objeto emissor ou receptor. No modelo com o *proxy*, ambas as portas do canal o conectam a dois diferentes *proxies*. Por exemplo, o Delay01 recebe as mensagens vindas do *proxy* do objeto emissor, e o Delay02 recebe as mensagens vindas do *proxy* do objeto receptor.

A condição para ambos os eventos é ter mensagens em suas respectivas filas, *Fila01* e *Fila02* .

A distribuição dos eventos Delay01 e Delay02 foi estudada em [16], onde dados foram coletados com o objetivo de analisar a distribuição do atraso na rede. Segundo [16], o atraso na rede se aproxima de uma distribuição normal truncada. Os testes feitos mostram que a média dos atrasos é igual a 329.35 e que o desvio padrão é igual a 80.32, baseando-se nisso foi obtido o coeficiente de variação igual a 0,24, de acordo com a fórmula a seguir:

$$coeficiente\_de\_variacao = \frac{desvio\_padrao}{media} \tag{5.3}$$

Uma vez sabido o valor do coeficiente de variação, é possível calcular a variância, parâmetro dos eventos Delay01 e Delay02. Trocando a ordem dos termos na fórmula 5.3, obtemos:

$$desvio\_padrao = (coeficiente\_de\_variacao) * (media) \tag{5.4}$$

E, sabendo que:

$$variancia = desvio\_padrao^2 \tag{5.5}$$

onde o coeficiente de variação é fixo e igual a 0,24 e a média é o atraso médio da rede, obtém-se a variância relativa à média do atraso da rede que se pretende simular.

5.3.4 Mensagens

As mensagens recebidas pelos objetos que simulam o emissor e os receptores são colocadas na fila *fila_msg* para futuro processamento. É verificado se a mensagem é um NACK. Caso seja, os limites mínimo e máximo do intervalo de execução do evento Enviar_Retransmissão são calculados de acordo com o emissor do NACK. O evento Tratar_Msg é clonado para tratamento da mensagem recebida.

As mensagens recebidas pela porta 1 dos objetos que simulam o canal são colocadas na fila *Fila01*. O evento Delay01 é clonado, com o objetivo de retirar a mensagem da fila e enviá-la para a porta 2, simulando um retardo sem espera de fila. O caminho inverso ocorre quando a mensagem é recebida pela porta 2.

O objeto *spliter* quando recebe uma mensagem simplesmente a envia para as portas diferentes da porta pela qual a mensagem foi recebida. Por exemplo, se a mensagem chegou no objeto através da porta 4, esta mensagem é enviada para as portas 1, 2 e 3.

As mensagens recebidas nos objetos *proxies* pela porta que o conecta ao canal, são repassadas somente para o objeto que representa a universidade e quando as mensagens são recebidas pela porta que o conecta à universidade, elas são repassadas para cada umas das portas que irão levá-las aos canais.

5.3.5 Recompensas

As recompensas criadas no modelo têm o objetivo de obter as seguintes medidas de interesse:

- perdas de dados e retransmissões;
- tempo de recuperação dos pacotes perdidos;
- número de NACKs enviados e recebidos por pacote perdido;
- número de retransmissões enviadas e recebidas por pacote perdido;
- número de pacotes de dados enviados pelo emissor;
- número de pacotes de dados recebidos pelos receptores;
- número de NACKs recebidos de um determinado objeto;
- número de retransmissões recebidas de um determinado objeto.

5.3.6 Variáveis de Estado

As variáveis de estado representam o estado interno dos objetos. No modelo, entre outras, podem ser encontradas as variáveis de estado *cache*, que mantém armazenadas informações sobre os dados recebidos, e *ret*, que mantém os *sns* a serem retransmitidos. As demais variáveis servem para controle das duas variáveis citadas anteriormente e para determinar a condição de execução de alguns eventos.

As declarações das variáveis, assim como comentários sobre elas, podem ser encontrados no Apêndice A

Capítulo 6

Análise de Desempenho

NESTE capítulo encontram-se os testes e as simulações realizadas com a finalidade de analisar o comportamento da transmissão Multicast confiável diante das diferentes situações propostas.

6.1 Testes

Após a implementação da biblioteca RML, vários experimentos na Internet foram realizados com o objetivo de testar o comportamento da transmissão Multicast confiável. Alguns experimentos são apresentados nesta dissertação com o objetivo de comparar seus resultados com os resultados obtidos através do modelo, e os demais experimentos pertencem ao estudo desenvolvido na dissertação [4].

6.1.1 Ambiente de Testes

A realização dos testes contou com a colaboração de quatro universidades, são elas: UFRJ (Universidade Federal do Rio de Janeiro), UFF (Universidade Federal Fluminense), UFJF (Universidade Federal de Juiz de Fora) e UMASS (Universidade de Massachusetts). Na Tabela 6.1 são descritas as características das máquinas utilizadas nos testes.

Universidade	**Máquina**	**Processador**	**Clock**	**Memória**
UFRJ	ilha	Pentium III	1 GHz	512 MB
UFF	abacate	2 x Pentium III	1 GHz	1 GB
UFJF	fenix	AMD - K6	200 MHz	64 MB
UMASS	newworld	Pentium III	1,266 GHz	2 GB

Tabela 6.1: Características das máquinas utilizadas nos testes

6.1.2 Proxy

Devido ao fato dos roteadores precisarem manter uma tabela com o estado de todos os membros do grupo Multicast, introduzindo complexidade e problemas de escalabilidade, poucos roteadores na Internet encontram-se com a transmissão Multicast habilitada. Portanto, para que a biblioteca RML pudesse ser testada, foi necessária a implementação de um *proxy*, denominado *rmproxy* (*Reliable Multicast Proxy*), cuja função é receber dados Multicast em uma rede local e repassá-los via conexão Unicast para os demais *proxies*, especificados em um arquivo de configuração. Além disso, receber dados, via conexão Unicast, e repassá-los para a rede local via conexão Multicast. Para não influenciar os resultados, o protocolo de transporte usado na conexão Unicast, estabelecida entre os *proxies*, é o UDP, portanto sem confiabilidade na transmissão, ou seja, podem ocorrer erros e perdas na transmissão, que deverão ser tratados pela biblioteca implementada. A Figura 6.1 ilustra o ambiente de teste com o uso do programa *rmproxy*.

6.1.3 Programa para Teste - rmtest

Para facilitar os testes, ao invés de mantermos usuários utilizando o TGWB em cada uma das universidades, o que seria inviável dispor de recursos humanos, foi implementado um programa, denominado *rmtest*, com a função de enviar e receber dados simulando uma apresentação no TGWB. Os dados são enviados segundo a análise feita no Capítulo 5 sobre as taxas de envio de dados em uma apresentação no TGWB. O programa *rmtest*, assim como o modelo, envia 25 pacotes com a probabilidade 0,7, 100 pacotes com a probabilidade 0,25 e 200 pacotes com a probabilidade

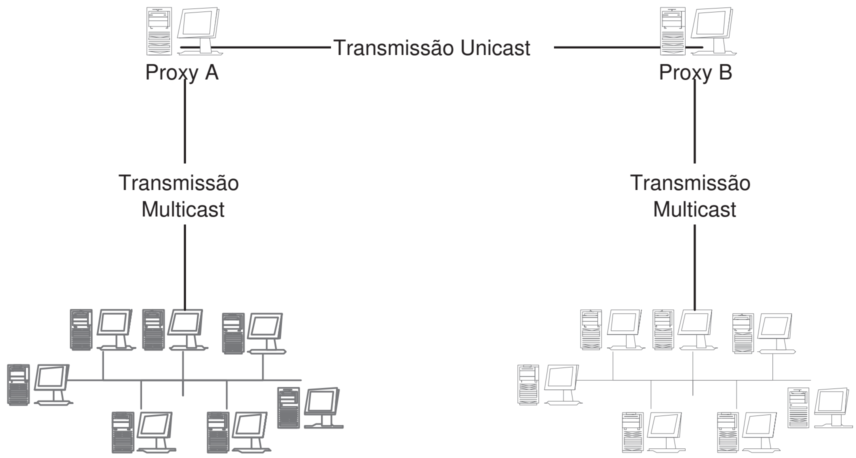

Figura 6.1: Ambiente de teste com o *rmproxy* (*Reliable Multicast Proxy*)

0,05.

Quando se deseja simular os testes feitos com o programa *rmtest*, é necessário obter alguns parâmetros dos testes para que a simulação tenha os resultados próximos dos resultados reais.

A implementação do *rmtest* viabilizou a flexibilidade nos testes, como a possibilidade da mudança de emissor, pois a cada momento uma universidade pode ser escalonada para ser a emissora dos dados. A realização dos testes pode ser feita a qualquer instante, bastando que as quatro máquinas especificadas na Tabela 6.1 estejam disponíveis.

O *rmproxy* e o *rmtest* podem estar ou não na mesma máquina. Nos testes e nas simulações foram considerados que ambos estão na mesma máquina, portanto o atraso e as perdas dos dados entre eles são nulos.

6.2 Simulações

As simulações foram feitas com diferentes cenários, tentando representar o máximo das variações encontradas em uma rede de transmissão de dados. Cada simulação foi feita com o intervalo de confiança de 95% e com 10 rodadas para aumentar a confiabilidade nos resultados.

Existem alguns parâmetros que devem ser inicializados quando se deseja simular o modelo, são eles:

- o tamanho da *cache*;
- o tamanho do vetor *ret*;
- as taxas de envio de dados;
- as constantes A, B, C, D, E e F;
- os temporizadores;
- os atrasos;
- as perdas.

A *cache* variou em 1000 e 4000 posições. O vetor *ret* variou em 10, 20, 50 e 255 posições, para análise do escalonamento das retransmissões. As taxas de envio de dados foram alteradas nas simulações de comparação do modelo com os testes, objetivando analisar a perda diante das diferentes taxas de envio. As constantes A, B, D, E e F variaram para a análise da influência destes parâmetros, porém a constante C se manteve igual a 5 devido ao fato deste parâmetro ter que ser necessariamente maior que 4, conforme explicado no Capítulo 5. O valor dos temporizadores, dos atrasos e das perdas variaram com o objetivo de analisar o comportamento da transmissão Multicast confiável de acordo com estas variações.

A simulação da transmissão Multicast através do modelo com o *spliter* requer uma análise do atraso e das perdas diferente da análise feita no modelo com o *proxy*. A disposição dos objetos é diferente, simulando diferentes ambientes na rede. Em seguida será explicado como foi feita a análise destes parâmetros em cada um dos modelos criados.

6.2.1 Modelo com o *spliter*

Atraso

O modelo (Figura 5.4) tem o valor do atraso nos canais de acordo com o valor obtido nos testes feitos com o programa *rmtest*. Contudo, somente são obtidos os atrasos entre cada uma das universidades e não o atraso entre a UFRJ e o *spliter*, sendo necessária a medição deste valor. Como o objeto *spliter* representa o último roteador comum entre a UFRJ e as demais universidades, foi medido o atraso entre a UFRJ e o roteador de endereço 200.20.94.58, identificado como o último roteador comum. Nas medições foi obtido o atraso de aproximadamente 5ms. Com isso, o valor do atraso no canal entre a UFRJ e o *spliter* não se alterou durante as simulações, mantendo-se em 5ms, e somente o valor dos atrasos entre o *spliter* e as universidades receptoras variou.

Para obter o valor do atraso no canal entre o *spliter* e as universidades receptoras, baseando-se no atraso obtido nos testes, que é o atraso total entre a UFRJ e as demais universidades e sabendo que o valor do atraso entre a UFRJ e o *spliter* é de aproximadamente 5ms, basta diminuir 5 ms do valor obtido nos testes. A diferença é o atraso considerado entre o *spliter* e as universidades receptoras. O valor do atraso nestes canais não é uma diferença direta como foi proposto nas simulações, porém, baseando-se no fato de que o atraso é instável, a diferença obtida de modo direto não influencia significativamente nos resultados das simulações. Se por exemplo, nos testes com o programa *rmtest* for obtido o atraso médio entre a UFRJ e a UFF de 100ms, então, no modelo é simulado o atraso médio de 5ms no canal entre a UFRJ e o *spliter* e de 95ms no canal entre o *spliter* e a UFF.

Estudo da Sensibilidade das Perdas

Assim como o atraso simulado no modelo é obtido através dos testes realizados, o mesmo acontece com as perdas. Com os resultados dos testes, é possível obter a perda entre a UFRJ e as demais universidades, porém, no modelo com o *spliter* existem dois canais entre cada uma das universidades. Entretanto, é preciso saber onde atribuir a perda obtida nos testes, uma vez que podem ocorrer perdas no canal entre a UFRJ e o *spliter* ou no canal entre o *spliter* e as universidades receptoras.

No caso do atraso, foram feitas medições de seu valor no canal entre a UFRJ e o roteador simulado pelo *spliter* e a diferença entre este valor e o valor total do atraso no canal entre a UFRJ e as universidades, foi atribuída ao canal entre o *spliter* e as universidades receptoras. Porém, devido à impossibilidade de acesso ao roteador para que pudesse ser medida a perda entre ele e a UFRJ, foi feito um estudo da sensibilidade da localização da perda.

O estudo da sensibilidade da perda consiste em analisar o comportamento do modelo diante das perdas nos canais. Para que o estudo pudesse ficar mais claro, os modelos foram ligeiramente modificados. Os eventos de espera por retransmissão, de envio de NACKs, de envio de retransmissões e de atraso no canal passaram a ter uma distribuição determinística, forçando os objetos a enviarem pacotes com a mesma taxa. As simulações descritas em seguida não fizeram uso dos parâmetros A, B, C, D, E e F, pois, para estas simulações, os eventos não possuem uma distribuição uniforme, onde estes parâmetros auxiliam no cálculo dos limites inferior e superior do intervalo do temporizador. Entretanto, estes eventos possuem uma distribuição determinística, conforme citado anteriormente. A Tabela 6.2 apresenta as taxas de envio associadas aos eventos.

Evento	**Taxa de envio**
Enviar_Nack	0,0005
Espera_Retransmissao	1
Enviar_Retransmissao	0,1
Delay01 e Delay02	1

Tabela 6.2: Taxas de envio associadas aos eventos determinísticos

As Figuras 6.2 e 6.3 mostram a distribuição das perdas nos dois modelos simulados para análise do estudo da sensibilidade. A perda simulada em ambos foi de 20%. No modelo da Figura 6.2, a perda está localizada entre a UFRJ e o *spliter*, portanto a perda sempre ocorre igualmente para as universidades receptoras, ou seja, quando ocorre uma perda entre a UFRJ e o *spliter*, todas as universidades receptoras perdem

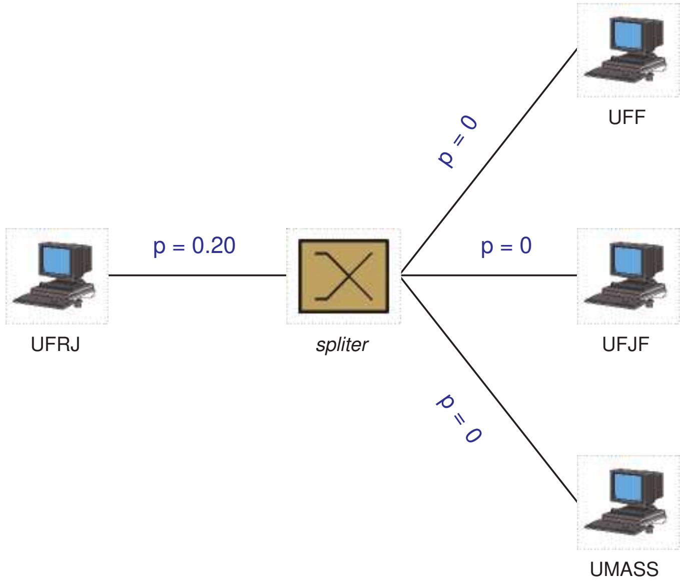

Figura 6.2: Modelo com a perda antes do *spliter*

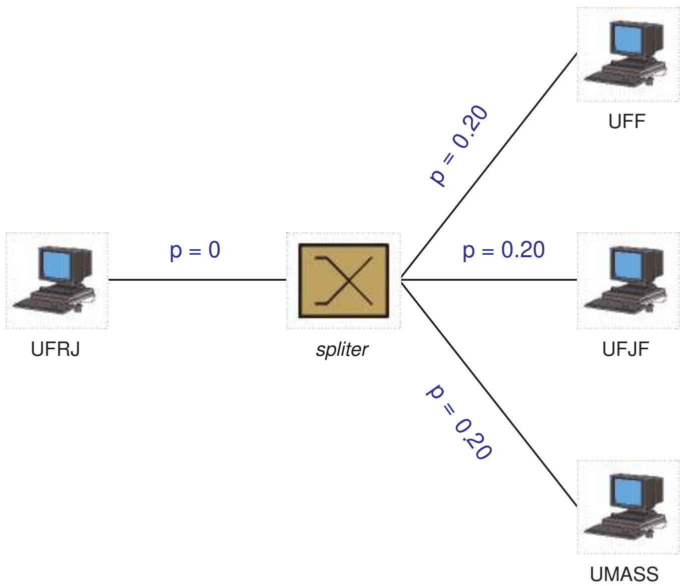

Figura 6.3: Modelo com a perda depois do *spliter*

o mesmo pacote. Com isso, somente a UFRJ pode retransmitir o pacote perdido e a probabilidade de sucesso na recuperação dos pacotes retransmitidos pela UFRJ é:

$$\text{Prob. de sucesso do modelo da Figura 6.2} = 1- \text{(probabilidade de falha)} = 1 - p$$

onde p é a probabilidade de perda.

No modelo da Figura 6.2, com $p = 0{,}20$, a probabilidade de sucesso é 0,80.

No modelo da Figura 6.3, a perda está localizada entre o *spliter* e as universidades receptoras. Como as perdas somente podem ocorrer depois do objeto *spliter*, não teremos neste caso perdas idênticas como no anterior, ou seja, quando ocorrer perdas, qualquer outro objeto estará habilitado para retransmitir. Portanto, a probabilidade de sucesso no recebimento dos pacotes é:

$$\begin{aligned}\text{Prob. de sucesso do modelo da Figura 6.3} &= 1- \text{(probabilidade de falha)}\\ &= 1- \text{(não receber retransmissão de nenhuma universidade)}\\ &= 1-(0+p)*(p+p)*(p+p)\\ &= 1-4p^3\end{aligned}$$

onde p é a probabilidade de perda.

No modelo da Figura 6.3, com $p = 0{,}20$, a probabilidade de sucesso é 0,968.

Para o modelo da Figura 6.3, temos uma probabilidade maior de sucesso no recebimento das retransmissões, ou seja, o tempo de recuperação é menor para este modelo quando comparado ao tempo de recuperação do modelo da Figura 6.2. As Tabelas 6.3 e 6.4 mostram a média do tempo de recuperação obtido na simulação dos dois modelos.

objeto	**UFF**	**UFJF**	**UMASS**
tempo médio de recuperação (ms)	496,4	496,4	496,4
intervalo de confiança (ms)	+/- 14,53	+/- 14,53	+/- 14,53

Tabela 6.3: Tempo de recuperação do modelo da Figura 6.2

objeto	UFF	UFJF	UMASS
tempo médio de recuperação (ms)	139	147	136,7
intervalo de confiança (ms)	+/- 5,46	+/- 4,02	+/- 3,70

Tabela 6.4: Tempo de recuperação do modelo da Figura 6.3

Nesta dissertação, em todas as simulações feitas com o modelo com o *spliter*, foi adotada a atribuição das perdas nos canais entre o *spliter* e as universidades receptoras, conforme proposto na Figura 6.3. Deste modo, torna-se mais fácil a inicialização do modelo, pois uma vez que as perdas nos canais entre a UFRJ e as universidades podem ser diferentes, basta colocar cada uma das probabilidades de perda nos canais entre o *spliter* e as respectivas universidades.

6.2.2 Modelo com o *proxy*

Atraso

Para modelar a transmissão Multicast confiável, simulando o uso do programa *rmproxy* feito nos testes, é necessário simular o valor do atraso entre cada uma das universidades. O modelo com o *spliter* poderia simular as conexões Unicast, propostas pelo *rmproxy*, se os atrasos entre elas fossem relativamente iguais. Porém, como isso não é verdade, foi necessária a criação do modelo com o *proxy*, que possui seis canais de comunicação simulando a conexão Unicast entre cada uma (Figura 5.5). A Tabela 6.5 mostra a média dos atrasos, obtidos através dos RTTs (*Round Trip Time*) coletados entre as universidades durante 15 dias, com intervalo de 1h entre o envio dos *pings*. Através desta tabela, além de observarmos que o valor do atraso entre elas é diferente, podemos detectar que o valor do atraso é um pouco diferente se muda o sentido da transmissão dos dados. Portanto, em cada canal foram simulados valores diferentes para os atrasos nos dois sentidos da transmissão e estes valores podem ser obtidos diretamente do teste a ser simulado ou da Tabela 6.5. Neste ponto, pode ser observado também, que o atraso em relação a UFF diminuiu quando comparado aos valores obtidos em testes anteriores.

Atraso	**UFRJ**	**UFF**	**UFJF**	**UMASS**
UFRJ	-	24,2ms	161,5ms	113,4ms
UFF	23,4ms	-	187,1ms	82,6ms
UFJF	149,9ms	168,9ms	-	219ms
UMASS	98,2ms	81,6ms	232,2ms	-

Tabela 6.5: Estimativa do atraso entre as universidades

Perdas

Através dos testes, somente podem ser obtidas as perdas entre a UFRJ e as demais. Portanto, para simular todas as perdas dos canais, durante uma transmissão Multicast, foi preciso obter um valor estimado das perdas entre cada uma das universidades. Foi feito um teste com baixa taxa de envio de dados, com a duração de 1h, onde apenas 1 ou 10 pacotes foram enviados, em intervalos entre 1 e 60s, simulando as retransmissões enviadas durante a transmissão Multicast. De acordo com estes testes, foram obtidas as probabilidades de perdas apresentadas na Tabela 6.6. As perdas do canal entre a UFRJ e as universidades receptoras são obtidas diretamente dos testes que se deseja simular, ou pode se considerar as médias das perdas obtidas em vários testes realizados ao longo do desenvolvimento desta dissertação, encontradas na Tabela 6.7.

Prob. de Perda	**UFRJ**	**UFF**	**UFJF**	**UMASS**
UFF	0,0562	-	0,0449	0,0787
UFJF	0,0212	0,0053	-	0,0212
UMASS	0,0058	0	0,1110	-

Tabela 6.6: Estimativa das perdas entre as universidades

Prob. de Perda	**UFF**	**UFJF**	**UMASS**
UFRJ	0,13	0,26	0,12

Tabela 6.7: Estimativa das perdas entre a UFRJ e as universidades

6.3 Análise dos Resultados

6.3.1 Comparando o Modelo com o *proxy* e os Testes Reais

Após a implementação do modelo, foi necessário testar se os resultados obtidos através do modelo descrevia a tendência do comportamento da biblioteca. Para isso, três testes foram escolhidos para serem simulados e terem seus resultados comparados. Ou seja, foi feito a comparação do comportamento da transmissão Multicast confiável a partir dos resultados obtidos do modelo com o *proxy* e dos resultados obtidos a partir dos testes feitos com a biblioteca RML. A Tabela 6.8 identifica cada dos testes que foram simulados. O parâmetro C se manteve igual a 5 para os dois primeiros testes e, igual a 7 no teste 3, de acordo com a exigência impostas no Capítulo 5.

Configurações	**Teste 1**	**Teste 2**	**Teste 3**
tamanho da *cache*	1000	1000	4000
número de pacotes enviados a cada intervalo entre 30 e 60s	25, 100 e 200	25 e 100	25, 50 e 100
constantes A, B, D, E e F	2	2	4

Tabela 6.8: Configurações dos testes que foram simulados

No teste 1 foram enviados 3775 pacotes, e as perdas obtidas foram 0,0448 entre a UFRJ e a UFF, 0,0445 entre a UFRJ e a UFJF e 0,0450 entre a UFRJ e a UMASS, e os atrasos obtidos no início dos testes encontram-se na Tabela 6.9. A Tabela 6.10 mostra os resultados deste teste e os Gráficos 6.4, 6.5 e 6.6 mostram o tempo de recuperação dos pacotes perdidos pelas universidades, de acordo com este teste.

Atraso	**UFRJ**	**UFF**	**UFJF**	**UMASS**
UFRJ	-	50ms	50ms	75ms
UFF	50ms	-	50ms	78ms
UFJF	50ms	50ms	-	101ms
UMASS	78ms	78ms	101ms	-

Tabela 6.9: Atrasos entre as universidades obtidos no teste 1

objeto	UFRJ	UFF	UFJF	UMASS
tempo médio de recuperação	-	2504ms	2278ms	2425ms
total de NACKs enviados	-	776	527	47
média de NACKs enviados por pacote	-	5,10	3,33	1,20
máximo de NACKs enviados por pacote	-	18	19	3
total de retrans. enviadas	533	26	27	21
média de retrans. enviadas por pacote	3,17	1	1	1
máximo de retrans. enviadas por pacote	16	1	1	1

Tabela 6.10: Resultados do teste 1

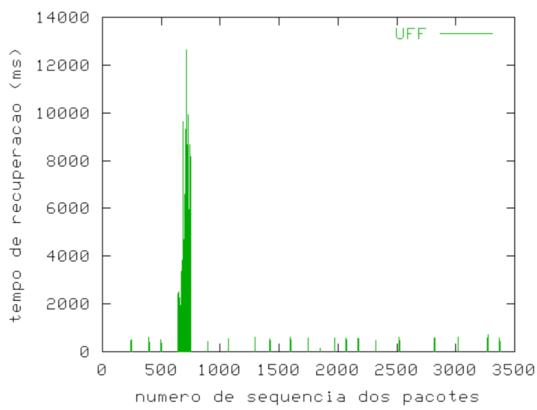

Figura 6.4: Tempo de recuperação da UFF obtido no teste 1

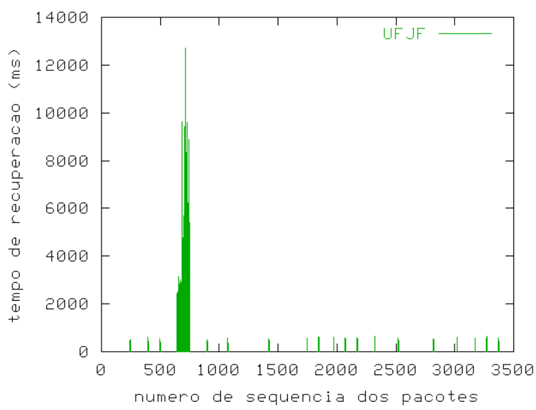

Figura 6.5: Tempo de recuperação da UFJF obtido no teste 1

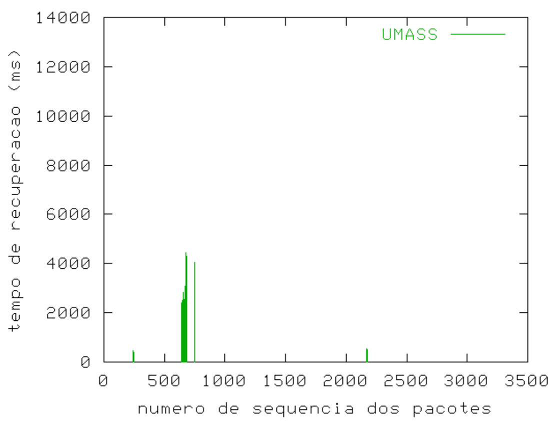

Figura 6.6: Tempo de recuperação da UMASS obtido no teste 1

A simulação do teste 1 considerou as perdas e os atrasos obtidos a partir deste teste. A Tabela 6.11 mostra os resultados da simulação do teste 1 e os Gráficos 6.7, 6.8 e 6.9 mostram o tempo de recuperação dos pacotes perdidos pelas universidades, de acordo com esta simulação. Pela simulação foram enviados 4077 pacotes.

objeto	UFRJ	UFF	UFJF	UMASS
tempo médio de recuperação	-	443ms	444ms	600ms
intervalo de confiança do tempo de recuperação	-	+/- 3,36ms	+/- 5,45ms	+/- 7,68ms
média de NACKs enviados	-	260	266,5	236
média de NACKs enviados por pacote	-	1,35	1,37	1,26
máximo de NACKs enviados por pacote	-	4	5	4
intervalo de confiança do número de NACKs enviados	-	+/- 0,015	+/- 0,016	+/- 0,020
média de retrans. enviadas	697	409	383	253
média de retransmissões enviados por pacote	1,26	1,19	1,14	1,07
máximo de retransmissões enviados por pacote	3	3	3	3
intervalo de confiança do número de retrans. enviadas	+/- 0,010	+/- 0,0084	+/- 0,013	+/- 0,0059

Tabela 6.11: Resultados da simulação do teste 1

O pico identificado nos Gráficos 6.4, 6.5 e 6.6 do tempo de recuperação do teste 1 é resultado do envio de uma rajada de 200 pacotes. Esta rajada ocasionou muitas perdas consecutivas e com isso o tempo médio de recuperação aumentou. Nas simulações, as perdas consideradas no canal são válidas durante toda a simulação, não simulando o envio com mais ou menos perdas para uma determinada rajada de dados. Devido a este fato, o tempo médio de recuperação dos pacotes, assim como o número de NACKs e retransmissões enviados no teste 1, é maior do que na simulação.

No teste 2, foram enviados 4000 pacotes, as perdas obtidas foram 0,0353 entre a UFRJ e a UFF, 0,0353 entre a UFRJ e a UFJF e 0,0351 entre a UFRJ e a UMASS e

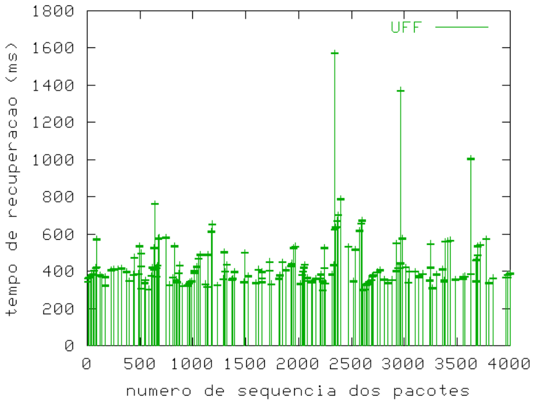

Figura 6.7: Tempo de Recuperação da UFF obtido na simulação do teste 1

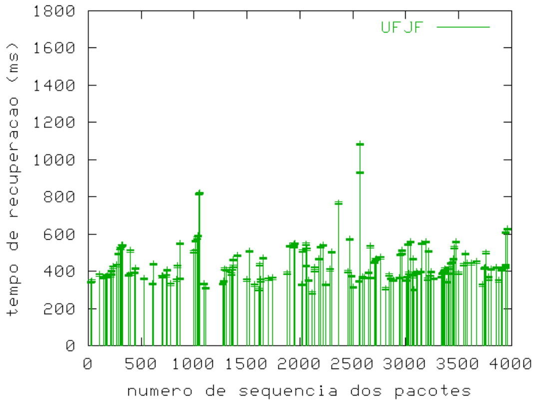

Figura 6.8: Tempo de Recuperação da UFJF obtido na simulação do teste 1

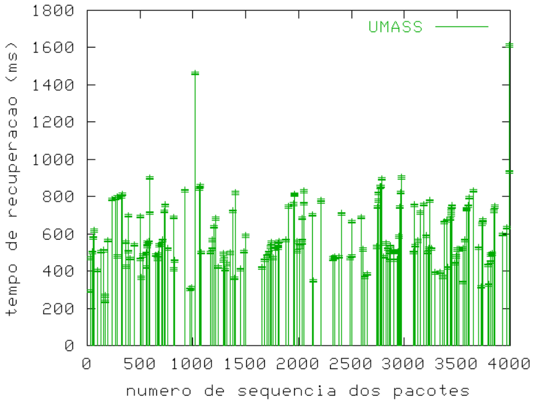

Figura 6.9: Tempo de Recuperação da UMASS obtido na simulação do teste 1

os atrasos obtidos no início dos testes estão na Tabela 6.12, que foram praticamente os mesmos atrasos obtidos no teste 1. Neste teste não ocorreu o envio de rajadas de 200 pacotes e, com isso, não houve muitas perdas consecutivas. A Tabela 6.13 mostra os resultados do teste 2 e os Gráficos 6.10, 6.11 e 6.12 mostram o tempo de recuperação dos pacotes perdidos pelas universidades, de acordo com este teste.

Atraso	**UFRJ**	**UFF**	**UFJF**	**UMASS**
UFRJ	-	50ms	50ms	75ms
UFF	50ms	-	50ms	78ms
UFJF	50ms	50ms	-	100ms
UMASS	78ms	78ms	100ms	-

Tabela 6.12: Atrasos entre as universidades obtidos no teste 2

objeto	UFRJ	UFF	UFJF	UMASS
tempo médio de recuperação	-	479ms	461ms	467ms
total de NACKs enviados	-	174	188	24
média de NACKs enviados por pacote	-	1,37	1,34	1
máximo de NACKs enviados por pacote	-	2	2	1
total de retrans. enviadas	211	18	19	12
média de retrans. enviadas por pacote	1,33	1	1	1
máximo de retrans. enviadas por pacote	2	1	1	1

Tabela 6.13: Resultados do teste 2

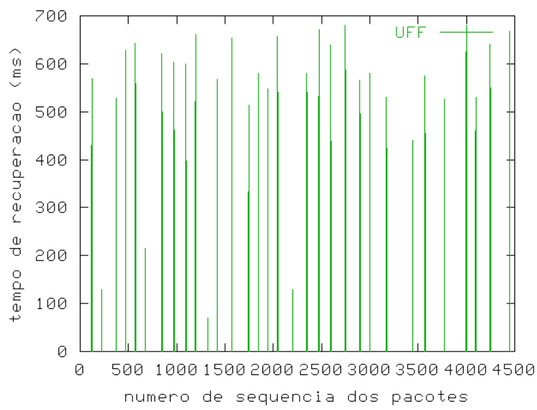

Figura 6.10: Tempo de recuperação da UFF obtido no teste 2

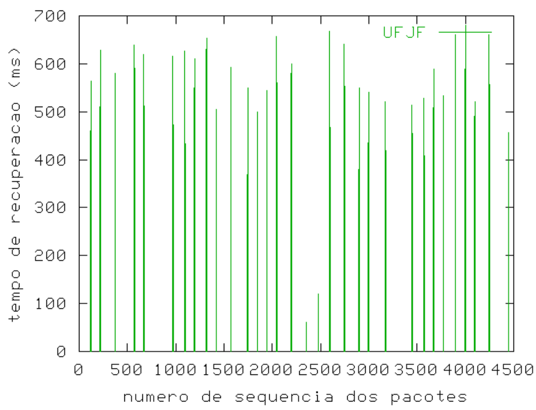

Figura 6.11: Tempo de recuperação da UFJF obtido no teste 2

A simulação do teste 2 considerou as perdas e os atrasos obtidos a partir deste teste. A Tabela 6.14 mostra os resultados da simulação e os Gráficos 6.13, 6.14 e

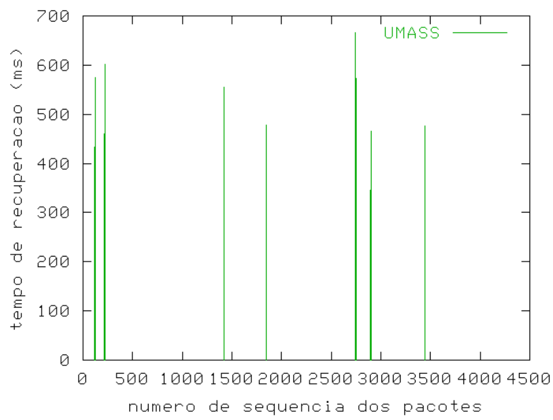

Figura 6.12: Tempo de recuperação da UMASS obtido no teste 2

6.15 mostram o tempo de recuperação dos pacotes perdidos pelas universidades, de acordo com esta simulação. Pela simulação foram enviados 3677 pacotes.

objeto	UFRJ	UFF	UFJF	UMASS
tempo médio de recuperação	-	426ms	426ms	565ms
intervalo de confiança do tempo de recuperação	-	+/- 5,47ms	+/- 3,12ms	+/- 4,24ms
média de NACKs enviados	-	190	172,8	151,7
média de NACKs enviados por pacote	-	1,30	1,30	1,17
máximo de NACKs enviados por pacote	-	3	3	3
intervalo de confiança do número de NACKs enviados	-	+/- 0,017	+/- 0,017	+/- 0,010
média de retrans. enviadas	465	282	272	189
média de retransmissões enviadas por pacote	1,20	1,15	1,12	1,06
máximo de retransmissões enviadas por pacote	3	3	3	3
intervalo de confiança do número de retrans. enviadas	+/- 0,004	+/- 0,012	+/- 0,005	+/- 0,009

Tabela 6.14: Resultados da simulação do teste 2

Como não houve rajadas de 200 pacotes no teste 2 e, consequentemente, não houve muitas perdas consecutivas, a simulação obteve resultados mais aproximados

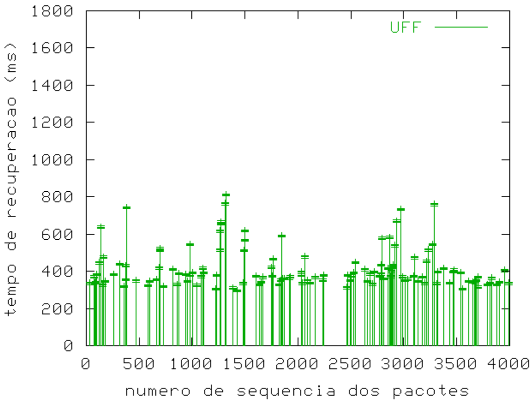

Figura 6.13: Tempo de recuperação da UFF obtido na simulação do teste 2

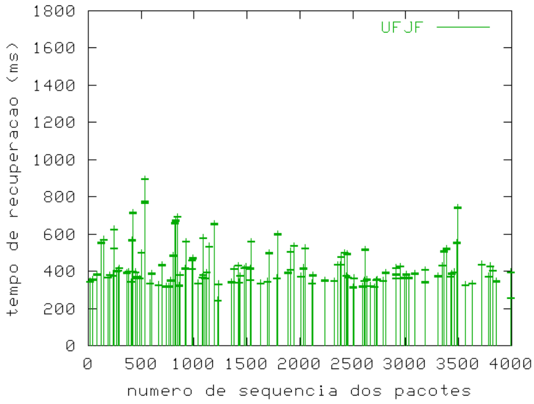

Figura 6.14: Tempo de recuperação da UFJF obtido na simulação do teste 2

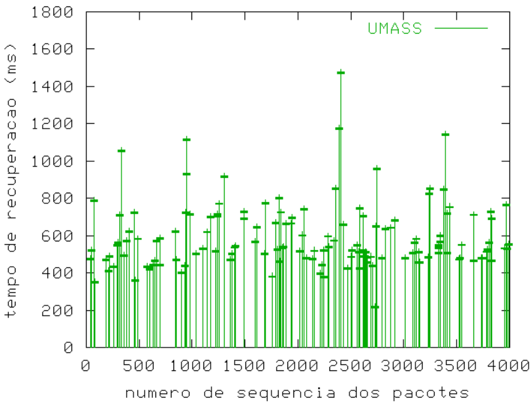

Figura 6.15: Tempo de recuperação da UMASS obtido na simulação do teste 2

dos resultados obtidos no teste 2.

A Tabela 6.16 mostra os resultados do teste 3 e os Gráficos 6.16, 6.17 e 6.18 mostram o tempo de recuperação dos pacotes perdidos pelas universidades, de acordo com este teste. Neste teste foram enviados 2900 pacotes, as perdas obtidas foram 0,0145 entre a UFRJ e a UFF, 0,2266 entre a UFRJ e a UFJF e 0,0152 entre a UFRJ e a UMASS, e os atrasos obtidos no início dos testes encontram-se na Tabela 6.15

Atraso	**UFRJ**	**UFF**	**UFJF**	**UMASS**
UFRJ	-	36ms	26ms	92ms
UFF	36ms	-	30ms	90ms
UFJF	42ms	24ms	-	115ms
UMASS	96ms	91ms	113ms	-

Tabela 6.15: Atrasos entre as universidades obtidos no teste 3

objeto	UFRJ	UFF	UFJF	UMASS
tempo médio de recuperação	-	802ms	4097ms	1299ms
total de NACKs enviados	-	71	4107	5
média de NACKs enviados por pacote	-	1,69	6,5	1
máximo de NACKs enviados por pacote	-	4	39	1
total de retrans. enviadas	742	612	2	346
média de retrans. enviadas por pacote	1,64	1,48	1	1,14
máximo de retrans. enviadas por pacote	5	4	1	3

Tabela 6.16: Resultados do teste 3

A simulação do teste 3 considerou as perdas e os atrasos obtidos a partir do mesmo. A Tabela 6.17 mostra os resultados da simulação e os Gráficos 6.19, 6.20 e 6.21 mostram o tempo de recuperação dos pacotes perdidos pelas universidades, de acordo com a simulação. Pela simulação foram enviados 2760 pacotes.

As universidades tiveram o comportamento semelhante quando comparamos o teste com a simulação. Devido à perda da UFJF ser maior que a perda das demais, ela teve o maior número de NACKs enviados. A UFRJ por possuir sempre todos os pacotes requisitados, teve o maior número de retransmissões enviadas, seguido da UFF, que teve a perda menor do que a UMASS, aumentando a sua chance de

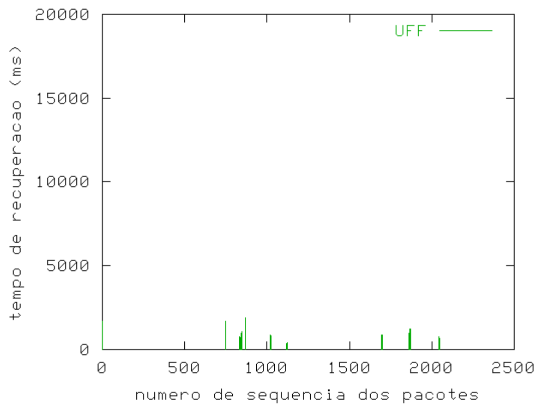

Figura 6.16: Tempo de recuperação da UFF obtido no teste 3

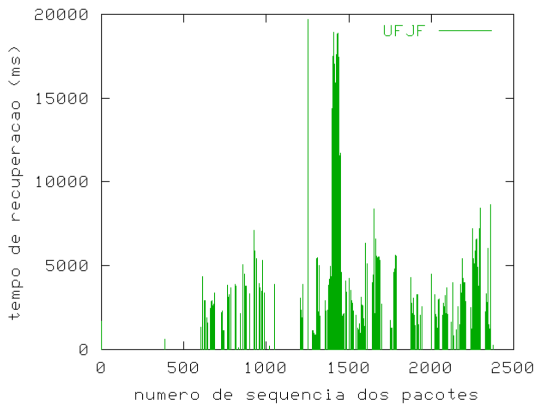

Figura 6.17: Tempo de recuperação da UFJF obtido no teste 3

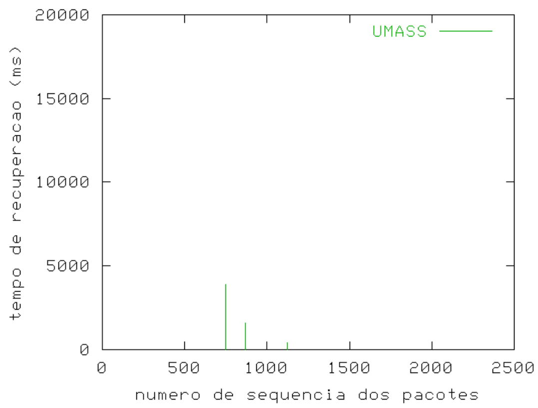

Figura 6.18: Tempo de recuperação da UMASS obtido no teste 3

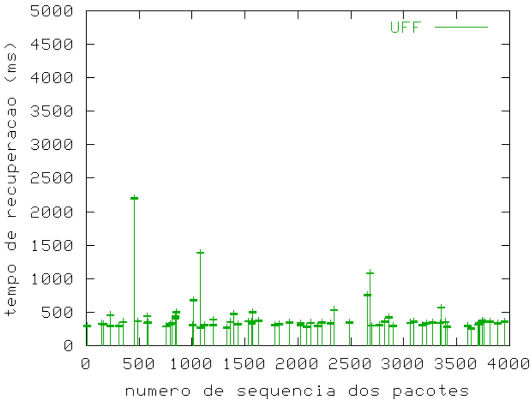

Figura 6.19: Tempo de recuperação da UFF obtido na simulação do teste 3

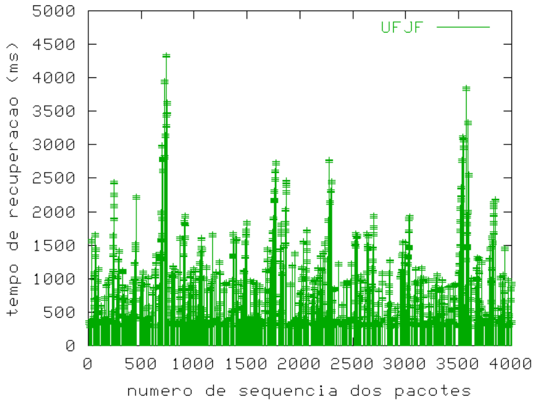

Figura 6.20: Tempo de recuperação da UFJF obtido na simulação do teste 3

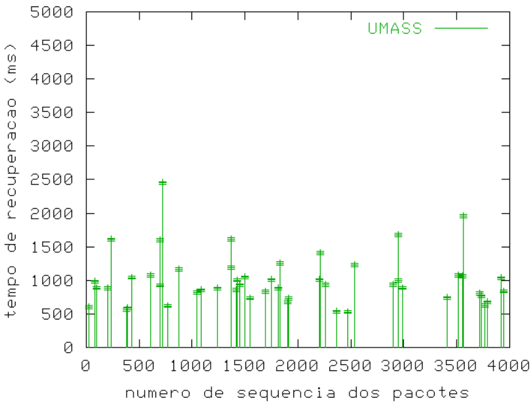

Figura 6.21: Tempo de recuperação da UMASS obtido na simulação do teste 3

objeto	UFRJ	UFF	UFJF	UMASS
tempo médio de recuperação	-	387ms	828ms	937ms
intervalo de confiança do tempo de recuperação	-	+/- 11,2ms	+/- 23,5ms	+/- 22,7ms
média de NACKs enviados	-	45	119	35
média de NACKs enviados por pacote	-	1,04	1,78	1,07
máximo de NACKs enviados por pacote	-	4	9	3
intervalo de confiança do número de NACKs enviados	-	+/- 0,012	+/- 0,040	+/- 0,015
média de retrans. enviadas	839,5	721,8	61,5	611,9
média de retransmissões enviadas por pacote	1,2	1,1	1	1,06
máximo de retransmissões enviadas por pacote	3	3	2	2
intervalo de confiança do número de retrans. enviadas	+/- 0,004	+/- 0,005	+/- 0,003	+/- 0,006

Tabela 6.17: Resultados da simulação do teste 3

ter o pacote requisitado para retransmissão. Quanto ao tempo de recuperação, no teste a UFJF teve o maior tempo e na simulação o maior tempo de recuperação foi o da UMASS. Como o atraso é maior nos canais que enviam dados a UMASS, nas simulações o tempo de recuperação ficou maior para esta. Porém, como a rede não é estável este comportamento não foi idêntico no teste 3, pois a rede pode ter parado por alguns instantes, resultando em um valor maior do tempo de recuperação para a UFJF. Este aumento no tempo de recuperação da UFJF pode ser percebido entre os pacotes de *sn* 1400 e 1500, vide Figura 6.17.

Podemos perceber que o modelo consegue simular a tendência do comportamento da transmissão Multicast na rede e, quanto menor for a variação do atraso durante a transmissão de dados, mais precisos serão os resultados da simulação.

6.3.2 Comparando o Modelo com o *spliter* e o Modelo com o *proxy*

Os dois modelos criados nesta dissertação possuem características diferentes. No modelo com o *spliter*, quando um pacote perdido é recuperado, a partir de alguma universidade receptora, a probabilidade de perda considerada no caminho entre a universidade que perdeu o dado e a universidade que vai retransmiti-lo é maior do que no modelo com o *proxy*. Isto acontece porque no modelo com o *spliter* a perda considerada nos canais é a perda obtida entre a UFRJ e as demais. Esta perda é obtida a partir dos testes, onde a UFRJ envia grandes quantidades de dados e quanto mais dados são enviados maior é a probabilidade de perda. De acordo com as simulações do modelo com o *spliter*, se o objeto UFF, por exemplo, retransmite um pacote para a UFJF, a probabilidade de perda no canal entre a UFF e o *spliter* é 0,13 e no canal entre o *spliter* e a UFJF é 0,26. Portanto, é considerada a perda entre a UFRJ e as demais e não a perda real entre a UFF e a UFJF, que segundo a Tabela 6.6 é menor.

No modelo com o *proxy*, a perda é calculada para cada canal. Como entre as universidades receptoras somente retransmissões são enviadas, a probabilidade de perda é menor quando comparada com as perdas do modelo com o *spliter*. De acordo com as simulações do modelo com o *proxy*, se o objeto UFF, por exemplo, retransmite um pacote para a UFJF, a probabilidade de perda no canal entre a UFF e a UFJF é de 0,0449 (vide Tabela 6.6). Portanto, devido às probabilidades de perda nos canais, o tempo de recuperação para o modelo com o *proxy* é menor do que no modelo com o *spliter*, como pode ser observado nos Gráficos 6.22, 6.23 e 6.24.

As perdas consideradas nas simulações são as perdas apresentadas nas Tabelas 6.6 e 6.7 e os atrasos em todos os canais é de 100ms.

A Tabela 6.18 apresenta o intervalo de confiança de 95% do tempo de recuperação das universidades para os dois modelos.

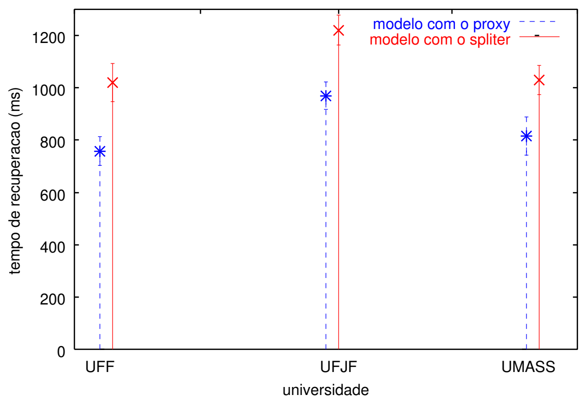

Figura 6.22: Tempo Médio de Recuperação

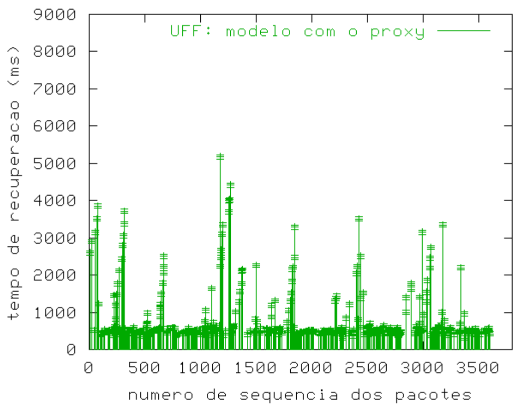

Figura 6.23: Tempo de recuperação por pacote da UFF no modelo com o *proxy*

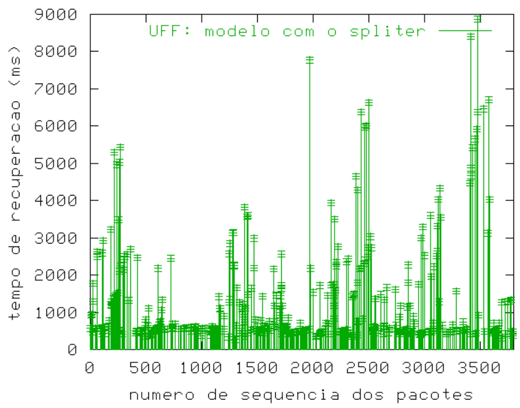

Figura 6.24: Tempo de recuperação por pacote da UFF no modelo com o *spliter*

Universidade	**Modelo com o *proxy***	**Modelo com o *spliter***
UFF	54,5572ms	72,9975ms
UFJF	53,4155ms	57,4156ms
UMASS	72,4091ms	56,8367ms

Tabela 6.18: Intervalo de confiança do tempo de recuperação

6.3.3 Análise da Variação dos Parâmetros A, B, D, E e F

Segundo as definições dos intervalos T_{NACK}, T_{WAIT} e T_{RET}, apresentados nos Capítulos 4 e 5, o aumento dos parâmetros A e B causa o aumento no intervalo de envio dos NACKs e o aumento dos parâmetros C e D causa o aumento do intervalo de espera por retransmissões, aumentando o tempo de recuperação dos pacotes. O aumento dos parâmetros E e F também causa o aumento no tempo de recuperação, pois o intervalo para envio das retransmissões é aumentado de acordo com estes dois parâmetros. As simulações apresentadas a seguir têm o objetivo de estudar o comportamento do tempo de recuperação, de acordo com as variações dos parâmetros. As simulações desta seção foram feitas com o modelo com o *proxy*.

Primeiramente, foram feitas duas simulações variando os valores dos parâmetros A, B, D, E e F com o objetivo de analisar o impacto na mudança de seus valores. O parâmetro C, em todas as simulações, variou de acordo com a exigência imposta no Capítulo 5. Em ambas as simulações foi assumido o atraso de 100ms em todos os canais e as perdas apresentadas nas Tabelas 6.6 e 6.7. Em uma das simulações o valor dos parâmetros A, B, D, E e F foi 2 e C = 5 e na outra o valor destes cinco primeiros parâmetros foi aumentado para 4 e C para 7.

Os Gráficos 6.25, 6.26 e 6.27 comparam o tempo de recuperação, o número de NACKs e retransmissões enviadas, respectivamente, obtidos nas simulações. O tempo de recuperação dos pacotes perdidos aumenta na simulação, onde A = B = D = E = F = 4, devido ao aumento do intervalo antes do envio dos NACKs. O número de NACKs também aumenta nesta simulação devido ao aumento do intervalo antes do envio das retransmissões, pois, como as retransmissões demoram mais a serem enviadas, o número de retransmissões diminui e, consequentemente, mais NACKs são enviados na rede.

Outra simulação foi feita com os seguintes valores: A = B = D = 4, E = F = 2 e C = 5. Mantendo o intervalo de tempo igual antes do envio das retransmissões, é possível analisar a diminuição do número de NACKs enviados (Figura 6.29), quando comparada com a simulação, onde A = B = D = E = F = 2, pois além do intervalo

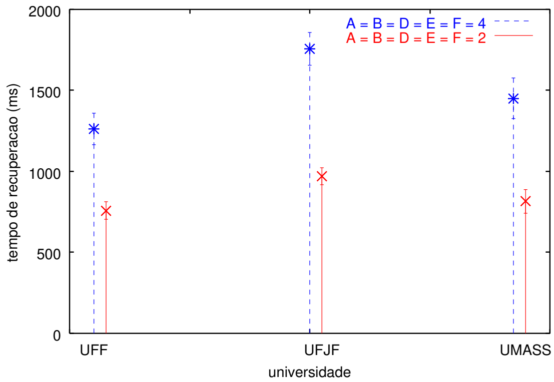

Figura 6.25: Tempo médio de recuperação

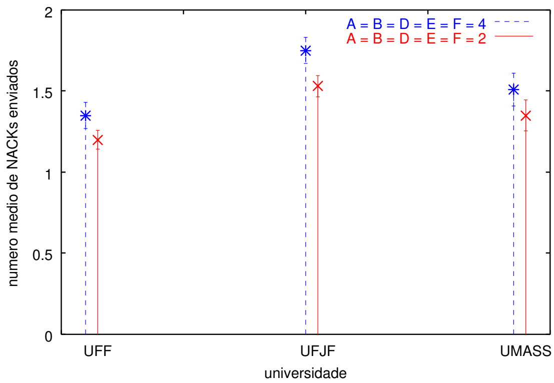

Figura 6.26: Número de NACKs enviados

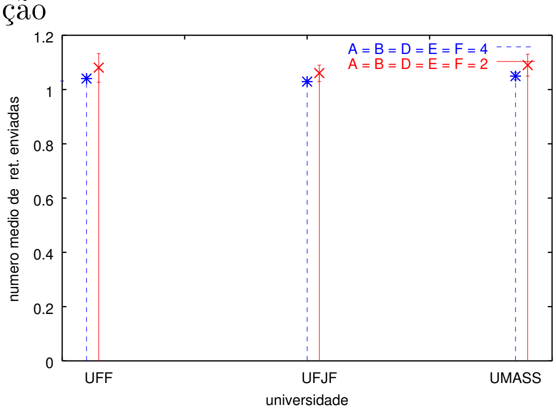

Figura 6.27: Número de retransmissões enviadas

para envio dos NACKs ter aumentado, as retransmissões continuam sendo enviadas com a mesma taxa. A comparação do tempo de recuperação pode ser observada na Figura 6.28. O número de retransmissões enviadas na simulação, onde A = B = D = 4 e E = F = 2, também diminuiu pois os NACKs demoram mais a serem enviados (Figura 6.30).

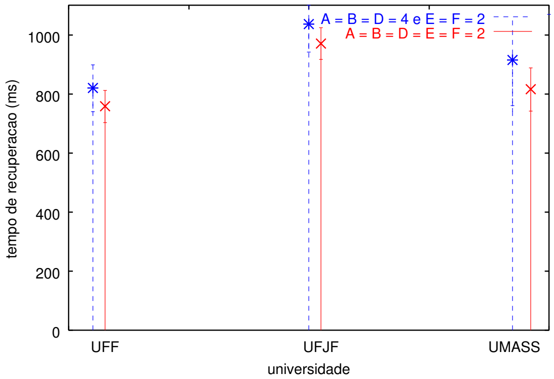

Figura 6.28: Tempo médio de recuperação, mantendo E e F = 2

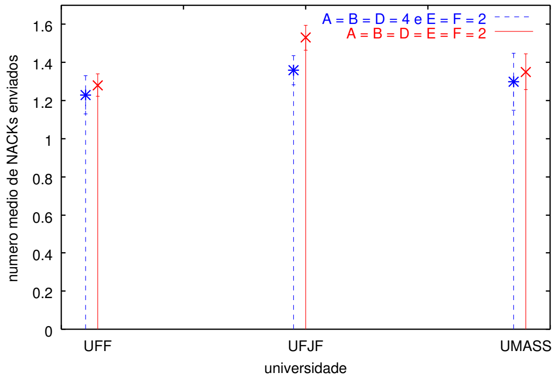

Figura 6.29: Número de NACKs enviados, mantendo E e F = 2

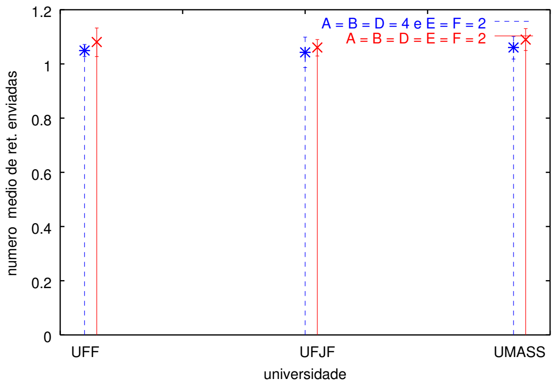

Figura 6.30: Número de retransmissões enviadas, mantendo E e F = 2

Na simulação com A = B = D = E = F = 4, além do tempo de recuperação aumentar muito em relação ao resultado usando os valores A = B = D = E = F = 2, o número de NACKs enviados também aumenta, portanto, não é eficiente a utilização do valor 4 para os parâmetros A, B, D, E e F. Na simulação com A = B = D = 4 e E = F = 2, o tempo de recuperação aumenta um pouco em relação ao resultado usando o valor 2 para os parâmetros A = B = D = E = F, mas o número de NACKs enviados é ligeiramente menor. Porém, se para a aplicação for primordial a recuperação rápida dos pacotes, não interessando se haverá mais NACKs enviados pela rede, é mais eficiente a utilização dos parâmetros A = B = D = E = F = 2, uma vez que na simulação com estes valores, o tempo de recuperação se apresentou menor em relação às outras duas simulações realizadas.

As Tabelas 6.19, 6.20 e 6.21 apresentam o intervalo de confiança de 95% do tempo de recuperação, do número de NACKs enviados e do número de retransmissões enviadas das três simulações estudadas nesta seção. Sendo a simulação 1, a simulação com A = B = D = E = F = 2; a simulação 2, com A = B = D = E = F = 4 e a simulação 3, com A = B = D = 4 e E = F = 2.

Simulação 1	**UFF**	**UFJF**	**UMASS**
tempo de recuperação (ms)	+/- 54,55	+/- 53,41	+/- 72,40
número de NACKs enviados	+/- 0,059	+/- 0,064	+/- 0,094
número de retransmissões enviadas	+/- 0,052	+/- 0,03	+/- 0,04

Tabela 6.19: Intervalo de confiança

Simulação 2	**UFF**	**UFJF**	**UMASS**
tempo de recuperação (ms)	+/- 97,44	+/- 101,42	+/- 126,62
número de NACKs enviados	+/- 0,082	+/- 0,080	+/- 0,1
número de retransmissões enviadas	+/- 0,008	+/- 0,007	+/- 0,013

Tabela 6.20: Intervalo de confiança

Simulação 3	**UFF**	**UFJF**	**UMASS**
tempo de recuperação (ms)	+/- 78,95	+/- 95,53	+/- 154,42
número de NACKs enviados	+/- 0,1	+/- 0,076	+/- 0,15
número de retransmissões enviadas	+/- 0,022	+/- 0,056	+/- 0,042

Tabela 6.21: Intervalo de confiança

6.3.4 Análise da Supressão dos NACKs e das Retransmissões

O Ambiente de Simulação

O ambiente criado no modelo para simular as supressões dos NACKs e das retransmissões é descrito a seguir:

- todos os canais com atraso de 100ms;
- a UFRJ considera o atraso de 10ms para cálculo do temporizador T_{RET};
- a UFJF considera o atraso de 200ms para cálculo do temporizador T_{NACK} e T_{WAIT}.

O temporizador T_{RET} da UFRJ possui um intervalo menor do que os intervalos dos temporizadores das demais universidades, uma vez que estes são calculados considerando o atraso de 100ms, definido nos canais. O fato do intervalo do temporizador T_{RET} da UFRJ ser menor aumenta a sua chance de enviar uma retransmissão que suprima as demais. Os temporizadores T_{NACK} e T_{WAIT} da UFJF, por serem calculados considerando o atraso de 200ms e não de 100ms como consideram as demais, tornam o intervalo de tempo antes do envio do NACK maior e, consequentemente, aumentam a chance de ter mais NACKs suprimidos. Estas diferenças em relação ao intervalo dos temporizadores entre as universidades foi feita para que a supressão dos NACKs e das retransmissões ficasse mais evidentes.

Variando a Perda

Foram feitas duas simulações, com o modelo com o *proxy*, para análise da supressão de NACKs e retransmissões em relação a variação das perdas, uma com as perdas apresentadas na Tabela 6.7 e a outra com as perdas de 40%, 60%, e 30% nos canais entre a UFRJ e a UFF, a UFRJ e a UFJF e a UFRJ e a UMASS, respectivamente. Ambas simulações consideram as perdas, apresentadas na Tabela 6.6 nos canais entre as universidades receptoras. Na simulação com as perdas obtidas a partir dos testes (Tabela 6.7) foram enviados em média 4220 pacotes de dados e na outra simulação 4140 pacotes. Ambas simulações com os parâmetros A, B, D, E e F com o valor 2 e C com o valor 5.

As simulações evidenciam que o aumento da perda leva ao aumento do número de NACKs e retransmissões enviados, como pode ser visto nas Tabelas 6.22 e 6.23. As perdas apresentadas nestas tabelas se referem aos canais entre a UFRJ e as universidades receptoras.

objeto	UFRJ	UFF	UFJF	UMASS
perda	-	13%	26%	12%
média de NACKs enviados	-	588	1159	522
média de NACKs enviados por pacote	-	1,067	1,090	1,068
máximo de NACKs enviados por pacote	-	4	19	3
média de retrans. enviadas	2132	1038	282	1127
média de retrans. enviadas por pacote	1,115	1,016	1,010	1,023
máximo de retrans. enviadas por pacote	4	2	2	2

Tabela 6.22: NACKs e retransmissões enviados na simulação com perdas obtidas a partir dos testes

objeto	UFRJ	UFF	UFJF	UMASS
perda	-	40%	60%	30%
média de NACKs enviados	-	2072	2101	1422
média de NACKs enviados por pacote	-	1,321	1,346	1,215
máximo de NACKs enviados por pacote	-	7	20	6
média de retrans. enviadas	5228	1675	770	1995
média de retrans. enviadas por pacote	1,512	1,045	1,022	1,072
máximo de retrans. enviadas por pacote	5	2	2	2

Tabela 6.23: NACKs e retransmissões enviados na simulação com perdas altas

A Tabela 6.24 mostra as percentagens, obtidas nas simulações, de NACKs e retransmissões suprimidos. Devido ao fato da UFRJ possuir o intervalo do temporizador T_{RET} com um valor significativamente inferior ao valor do mesmo temporizador nas demais universidades, ela não teve nenhuma retransmissão suprimida. Como a UFJF possui a maior perda, menos pacotes são escalonados para retransmissão, com isso, a quantidade de retransmissões suprimidas representa uma maior percentagem.

Por exemplo, se a UFJF possui três retransmissões escalonadas e a UMASS possui dez, e a UFRJ envia uma retransmissão que suprime uma retransmissão em cada uma dessas universidades, então, são suprimidas aproximadamente 33% das retransmissões escalonadas na UFJF e 10% na UMASS. A UFJF, por ter o intervalo dos temporizadores T_{NACK} e T_{WAIT} maior, teve a maior percentagem de supressão de NACKs.

objeto	UFF	UFJF	UMASS	UFF	UFJF	UMASS
perda	13%	26%	12%	40%	60%	30%
NACKs suprimidos	49%	50%	48%	44%	54%	46%
retrans. suprimidas	31%	65%	30%	40%	50%	39%

Tabela 6.24: NACKs e retransmissões suprimidos

Variando os Temporizadores

Outras duas simulações foram feitas para análise da supressão de NACKs e retransmissões em relação a variação dos temporizadores. Ambas consideraram as perdas apresentadas na Tabela 6.7 nos canais entre a UFRJ e a UFF, a UFRJ e a UFJF e a UFRJ e a UMASS, respectivamente e as perdas, apresentadas na Tabela 6.6 nos canais entre as universidades receptoras. Em uma simulação os parâmetros A, B, D, E e F teve o valor 4 e C o valor 7 e na outra, A, B, D, E e F teve o valor 6 e C o valor 9. Com isso, foi possível analisar a sensibilidade da supressão de pacotes em relação ao aumento dos temporizadores.

Os membros que possuem o menor atraso em relação ao emissor devem enviar NACKs e retransmissões antes, estes podem suprimir os NACKs e as retransmissões dos membros com atraso maior. Portanto, quanto mais distante um membro estiver do emissor e dos outros membros, maior a chance de supressão para os pacotes enviados por este membro. No cenário simulado, a UFJF possui o maior atraso em relação ao emissor, portanto, esta foi escolhida para a análise da sensibilidade da supressão de pacotes em relação a variação dos temporizadores.

Com o aumento dos parâmetros, o intervalo antes do envio dos NACKs aumenta mais para a UFJF do que para as demais, pois o seu atraso em relação ao emissor é maior, podendo com isso aumentar a chance de um NACK escalonado por esta ser suprimido. O mesmo comportamento ocorre em relação a supressão de retransmissão. Como o atraso considerado para cálculo do temporizador T_{RET} da UFRJ é 10ms, seu intervalo de tempo antes de enviar uma retransmissão é menor, mesmo com o aumento dos parâmetros. Com isso, pode aumentar a chance da UFRJ suprimir as retransmissões das demais universidade, ou seja, pode aumentar a chance da UFJF, por exemplo, ter retransmissões suprimidas conforme o aumento dos temporizadores. Porém, a Tabela 6.25 mostra que a percentagem de supressão de pacotes da UFJF aumenta pouco conforme a variação dos parâmetros dos temporizadores. Portanto, segundo evidências obtidas através do modelo, a supressão dos NACKs e das retransmissões é pouco sensível em relação ao aumento dos temporizadores.

valor do parâmetro C	4	7	9
NACKs suprimidos	50%	51,8%	52%
retrans. suprimidas	65%	66,2%	66,8%

Tabela 6.25: NACKs e retransmissões suprimidos da UFJF, conforme aumento dos temporizadores

6.3.5 Análise da Variação dos Atrasos, dos Temporizadores e das Perdas

Esta seção tem o objetivo de analisar o comportamento da transmissão Multicast de acordo com as variações dos atrasos, dos temporizadores e das perdas. Para isto, foram feitas 63 simulações com o modelo com o *proxy*, considerando as perdas da Tabela 6.6 para os canais entre as universidades receptoras e variando a perda em 10%, 20% e 30% no canal entre a UFRJ e as universidades. Os atrasos variaram em 100ms, 300ms e 500ms. Os temporizadores T_{NACK}, T_{WAIT} e T_{RET} foram calculados segundo as variações do atraso em 100ms, 150ms, 200ms, 250ms, 300ms, 350ms e 400ms. Portanto, os valores destes temporizadores (em *ms*) variaram segundo os intervalos apresentados a seguir:

- T_{NACK}: [200,400], [300,600], [400,800], [500,1000], [600,1200], [700,1400], [800,1600];
- T_{WAIT}: [500,700], [750,1050], [1000, 1400], [1250,1750], [1500,2450], [2000,2800];
- T_{RET}: [200,400], [300,600], [400,800], [500,1000], [600,1200], [700,1400], [800,1600].

Nos gráficos apresentados nesta seção, o eixo x indica o valor mínimo do intervalo dos temporizadores T_{NACK} e T_{RET}. Os seguintes parâmetros se mantiveram constantes durante estas simulações:

- tamanho da *cache* = 4000;
- número de pacotes enviados a cada intervalo entre 30 e 60s = 25, 100 e 200;
- constantes A, B, D, E e F = 2 e C = 5.

Os Gráficos 6.31, 6.32, 6.33, 6.34, 6.35 e 6.36 apresentam o tempo de recuperação (em *ms*) da UFF e o número de NACKs enviados por pacote pela UFF para os atrasos nos canais de 100ms, 300ms e 500ms. Como os resultados do tempo de recuperação e do número de NACKs enviados pelas universidades foram semelhantes, somente os resultados da UFF serão apresentados nesta dissertação. Nestes gráficos, podemos perceber que o aumento da perda ocasiona o aumento do tempo de recuperação e, o número médio de NACKs enviados pouco variou (note que os intervalos de confiança se sobrepõem). Além disso, ao compararmos os gráficos com os atrasos nos canais de 100ms, 300ms e 500ms, percebemos que o tempo de recuperação aumenta proporcionalmente ao aumento do atraso no canal. Porém, o número médio de NACKs enviados pouco variou, ou seja, comparando os mesmos valores dos temporizadores para diferentes atrasos nos canais, o número médio de NACKs enviados pouco varia, mostrando a pouca sensibilidade do número de NACKs enviados em relação ao aumento do atraso no canal.

O aumento do intervalo dos temporizadores gera o aumento, praticamente linear, do tempo de recuperação, porém, o número médio de NACKs enviados se manteve praticamente constante (vide Gráficos 6.34, 6.35 e 6.36). Por exemplo, o Gráfico 6.35, onde o número médio de NACKs enviados por pacote se manteve na faixa entre 1,186 e 1,615 para a perda de 10%, 1,510 e 1,725 para a perda de 20% e 1,766 e 1,936 para a perda de 30%.

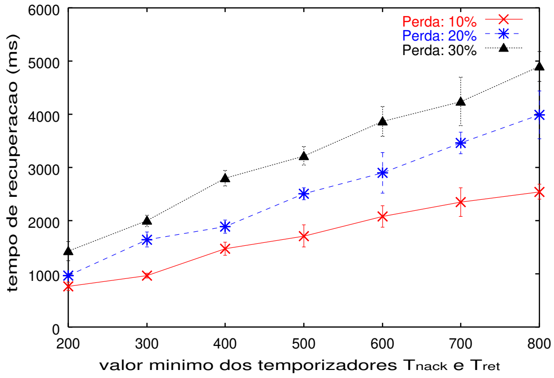

Figura 6.31: Tempo médio de recuperação da UFF com o atraso de 100ms nos canais

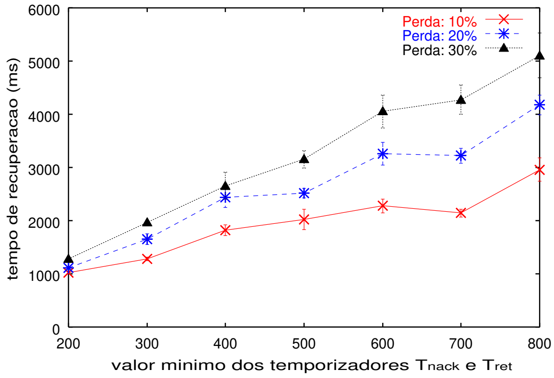

Figura 6.32: Tempo médio de recuperação da UFF com o atraso de 300ms nos canais

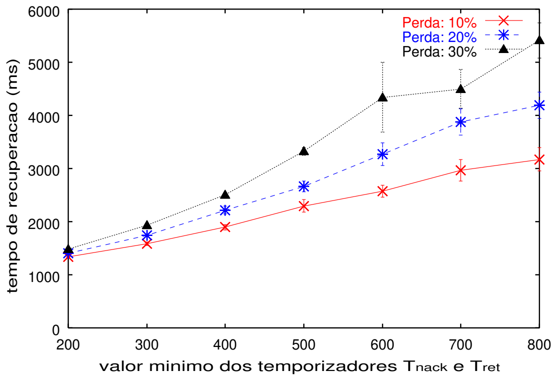

Figura 6.33: Tempo médio de recuperação da UFF com o atraso de 500ms nos canais

Os Gráficos 6.37, 6.38 e 6.39 apresentam o número de retransmissões enviadas por pacote pela UFRJ para os atrasos nos canais de 100ms, 300ms e 500ms. Em cada gráfico, podemos perceber que o aumento da perda ocasiona o aumento do número de retransmissões enviadas. O número de retransmissões enviadas se manteve praticamente constante para a simulação com o atraso no canal de 100ms. Nas simulações com o atraso no canal de 300ms e 500ms pode ser observado que o número médio de retransmissões enviadas por pacote diminui conforme aumenta o intervalo dos temporizadores. Esta diminuição é mais evidente no gráfico para atraso de

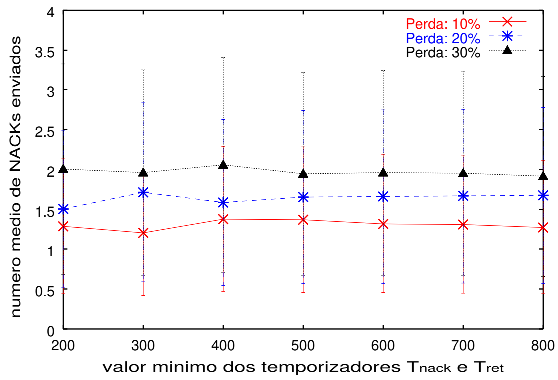

Figura 6.34: Número médio de NACKs enviados por pacote pela UFF com o atraso de 100ms nos canais

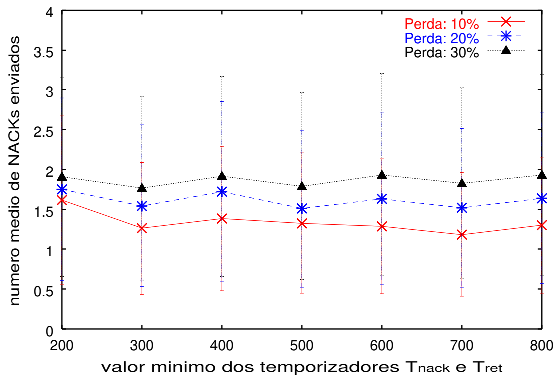

Figura 6.35: Número médio de NACKs enviados por pacote pela UFF com o atraso de 300ms nos canais

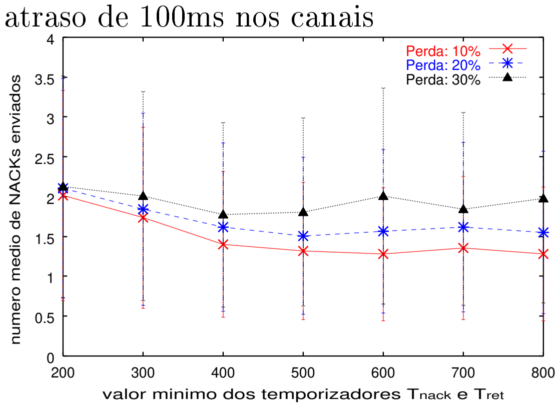

Figura 6.36: Número médio de NACKs enviados por pacote pela UFF com o atraso de 500ms nos canais

500ms, pois quanto maior é o atraso no canal, maior é o intervalo de chegada entre os NACKs, com isso, o aumento do intervalo dos temporizadores antes do envio das retransmissões aumenta a probabilidade de uma única retransmissão valer para um maior número de NACKs recebidos, sem que haja a necessidade de retransmissões repetidas e, consequentemente, diminuindo o número de retransmissões enviadas. Portanto, conforme aumenta o atraso no canal, a variação do número de retransmissões torna-se mais sensível à variação do intervalo dos temporizadores. Porém, como o atraso no canal geralmente é menor que 500ms (vide Tabela 6.5), o número de retransmissões tende a não variar muito diante das variações dos temporizadores.

Devido ao número médio de NACKs enviados variar pouco em relação ao aumento do atraso no canal, temos, como consequência, uma pequena variação do número de retransmissões enviadas em relação ao aumento deste atraso. Este comportamento é evidenciado quando comparamos o número médio de retransmissões enviadas apresentado nos Gráficos 6.37, 6.38 e 6.39.

Alguns gráficos apresentaram um comportamento não esperado, como o Gráfico 6.36, onde o número de NACKs enviados para o intervalo mínimo do temporizador em 600ms e perda de 30% é maior do que para o intervalo mínimo do temporizador em 500ms e mesma perda. Entretanto, este número deveria ser menor, pois uma vez que o intervalo para envio dos NACKs aumenta, menos NACKs deveriam ser enviados pela rede. Porém, as simulações foram rodadas simulando 1h de apresentação no TGWB e, uma possível explicação para esta aparente incoerência é que este tempo não é suficiente para que o modelo entre em estado estacionário. Para verificar esta suposição, o modelo com o intervalo mínimo do temporizador em 600ms e perda de 30% foi rodado novamente simulando 5h de apresentação no TGWB. Com os novos resultados, foi verificado que o número de NACKs diminuiu em relação ao número de NACKs do intervalo mínimo do temporizador em 500ms, conforme o esperado e, esta diminuição pode ser observada no Gráfico 6.40.

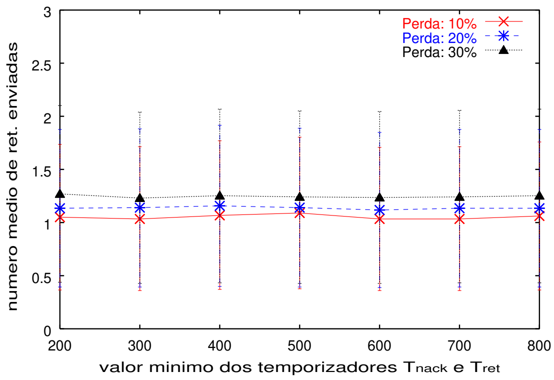

Figura 6.37: Número médio de retransmissões enviadas por pacote pela UFRJ com o atraso de 100ms nos canais

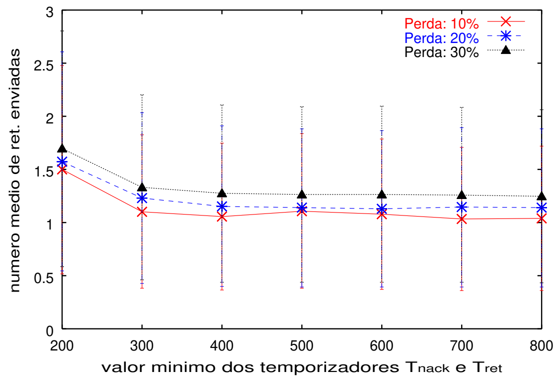

Figura 6.38: Número médio de retransmissões enviadas por pacote pela UFRJ com o atraso de 300ms nos canais

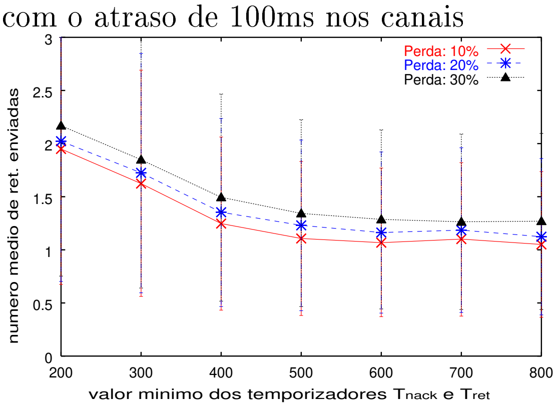

Figura 6.39: Número médio de retransmissões enviadas por pacote pela UFRJ com o atraso de 500ms nos canais

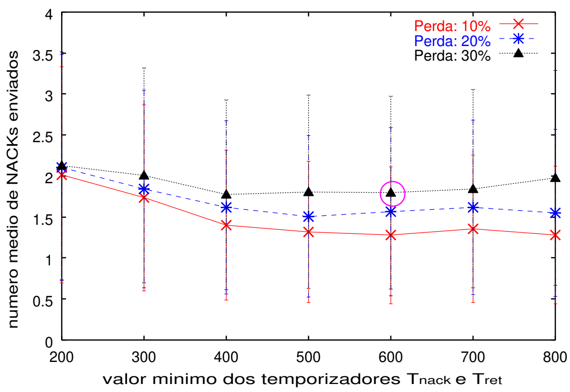

Figura 6.40: Número médio de NACKs enviados pela UFF com o atraso de 500ms nos canais, simulando 5h de apresentação no TGWB

Conclusão

Com o aumento do intervalo dos temporizadores, o número médio de NACKs e retransmissões enviadas se mantém praticamente constantes, além do tempo de recuperação aumentar sensivelmente. Portanto, para os cenários estudados, as simulações evidenciam que o aumento do intervalo dos temporizadores não é recomendável. Com isso, podemos concluir que o valor deste intervalo deve ser o menor possível, visando um menor tempo de recuperação dos pacotes perdidos, porém, evidentemente, garantindo a supressão de NACKs e de retransmissões.

Capítulo 7

Conclusão

As redes de computadores, em particular, as aplicações multimídia distribuídas, continuam a desenvolver-se em um ritmo acelerado, crescendo em tamanho e complexidade. Com isso, torna-se cada vez mais importante ter processamento e largura de banda suficientes para a distribuição desses tipos de dados.

A transmissão Multicast pode reduzir o número de pacotes duplicados no canal, economizando largura de banda e, conseqüentemente, otimizando o uso da rede. Como a transmissão Multicast não é confiável, foi necessária a realização de um trabalho para garantir a confiabilidade desta transmissão. Portanto, o estudo desta dissertação concentra-se na implementação da confiabilidade e na análise de desempenho da transmissão Multicast.

Após os estudos realizados no Capítulo 2 sobre os tipos de protocolos para transmissão Multicast confiável, onde foram analisadas as vantagens e as desvantagens de cada um deles, o protocolo Enhanced-Receiver-Initiated foi escolhido para ser implementado, devido à superioridade de seu desempenho em relação aos demais protocolos. Com isso fica atribuído ao receptor a responsabilidade da recuperação dos dados perdidos.

Para garantia da confiabilidade na transmissão Multicast, foi implementada, em equipe, a biblioteca denominada *Reliable Multicast Library* (RML), que pode ser vista como um protocolo em uma camada superior à camada de transporte, com a função de detectar perdas de pacotes e recuperar eficientemente os pacotes perdidos, tornando a transmissão Multicast confiável.

O desenvolvimento da ferramenta TGWB já havia sido iniciado por outros alunos, porém faltava garantir a confiabilidade na transmissão Multicast. Portanto, o TGWB foi integrado a biblioteca RML, como resultado de um trabalho em equipe, envolvendo mais dois alunos, um de mestrado e outro de graduação. No momento desta integração algumas funcionalidades do TGWB foram aprimoradas. O TGWB foi a principal aplicação distribuída que fez uso da biblioteca para garantir a confiabilidade na transmissão Multicast. As demais aplicações usadas, como o programa *rmtest*, testaram a biblioteca simulando o funcionamento do TGWB.

Também foram criados dois modelos da transmissão Multicast confiável simulando diferentes ambientes na rede. Primeiramente, foi implementado um modelo mais genérico, onde se considera que toda a comunicação é feita através da transmissão Multicast. Em seguida, foi implementado um modelo que representa o ambiente de testes criado para validar o comportamento do TGWB com o uso da biblioteca RML desenvolvida.

A análise de desempenho feita no Capítulo 6 mostrou que a transmissão Multicast se comporta de diferentes maneiras, dependendo da situação da rede no momento da transmissão. A possibilidade de mudar os parâmetros do modelo para analisar diferentes situações torna o estudo do comportamento da transmissão Multicast confiável prático e motivador.

Como resultados deste trabalho estão disponibilizados, em domínio público [1], a biblioteca RML, que garante a confiabilidade em uma transmissão Multicast, e o modelo que a descreve, tornando possível à análise de desempenho e o ajuste de parâmetros para diferentes condições da Internet. Além disso, a distribuição internacional da ferramenta TGWB.

7.1 Trabalhos Futuros

Duas linhas de atuação podem estender o trabalho aqui apresentado: os aperfeiçoamentos na biblioteca RML, com o objetivo de otimizar o trabalho desenvolvido e as modificações no modelo implementado.

Na biblioteca RML podem ser acrescentadas algumas funcionalidades, como:

- otimização do cálculo dos temporizadores, fazendo com que o cálculo seja feito dinamicamente. Com isso, o intervalo do temporizador estará sempre atualizado de acordo com as variações do atraso na rede;
- prevenção de NACKs prematuros, ou seja, aumentar gradualmente o intervalo de envio de NACKs para o mesmo pacote perdido;
- ignorar pedidos duplicados de retransmissões, ou seja, não retransmitir pacotes que acabaram de ser retransmitidos.

O acréscimo destas funcionalidades visariam a diminuir o tempo de recuperação dos pacotes perdidos e ao número de pacotes NACKs e retransmissões enviados pela rede.

O modelo poderia ser modificado para abranger, além dos dois ambientes implementados nesta dissertação, mais ambientes de transmissão Multicast. Nesta modificação pode estar incluída o acréscimo de novos objetos, como outros roteadores e membros. E, caso fossem implementadas as funcionalidades anteriormente sugeridas para a biblioteca, estas poderiam ser facilmente implementadas no modelo.

Outro trabalho que poderia ser feito, é a implementação do Multicast na aplicação, possibilitando o uso de aplicações distribuídas em redes sem suporte ao Multicast nativo.

Bibliografia

[1] URL http://www.land.ufrj.br/. Site do LAND.

[2] URL http://www.linuxdoc.org/howto/multicast-howto-2.html/. Site do Linux.

[3] URL http://www.microsoft.com/windows/netmeeting/features/whiteboard/default.asp. Site do NetMeeting.

[4] Azevedo, J. A. Transmissão Multicast Confiável: Aplicação na Ferramenta Tangram Whiteboard e Experimentos na Internet. Tese de Mestrado, NCE/UFRJ, 2002.

[5] Barbosa, V. C. *An introduction to distributed algorithms.* MIT Press, 1996.

[6] Berson, S., de Souza e Silva, E., e Muntz, R. R. *Numerical Solution of Markov Chains.* Marcel Dekker, Inc., 1991, ch. An Object Oriented Methodology for the Specification of Markov Models, pp. 11–36.

[7] Brian Neil Levine and David B. Lavo and J. J. Garcia-Luna-Aceves. The case for reliable concurrent multicasting using shared ack trees. In *ACM Multimedia* (1996), pp. 365–376.

[8] Cardozo, A. Q. Mecanismos para Garantir a Qualidade de Serviço de Aplicações de Vídeo sob Demanda. Tese de Mestrado, COPPE/UFRJ, 2002.

[9] Carmo, R., de Carvalho, L., de Souza e Silva, E., Diniz, M., e Muntz, R. Performance/Availability Modeling with the TANGRAM-II Modeling Environment. *Performance Evaluation 33* (1998), 45–65.

[10] Chawathe, Y. *An Architecture for Internet Broadcast Distribution as an Infrastructure*. PhD thesis, 2000.

[11] Cheng, W. C. URL http://bourbon.cs.umd.edu:8001/tgif/. TANGRAM Graphic Interface Facility.

[12] Chu, Y. H., Rao, S. G., , Seshan, S., e Zhang, H. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *ACM SIGCOMM 2001* (California, USA, 2001), ACM, pp. 27–31.

[13] Chu, Y. H., Rao, S. G., e Zhang, H. A Case for End System Multicast. In *ACM SIGMETRICS 2000* (Santa Clara, CA, junho de 2000), ACM, pp. 1–12.

[14] de Brito, C. E. F., de Moraes, R. S., de Oliveira, D. J., e de Souza e Silva, E. Comunicação Multicast Confiável na Implementação de uma Ferramenta Whiteboard. In *XVII Simpósio Brasileiro de Redes de Computadores* (1999), pp. 222–237.

[15] de Souza e Silva, E., e Muntz, R. R. *Métodos Computacionais de Solução de Cadeias de Markov: Aplicações a Sistemas de Computação de Comunicação.* VIII Escola de Computação - Gramado, 1992, pp. 1–13.

[16] Elteto, T., e Molnar, S. On the Distribution of Round-Trip Delays in TCP/IP Networks. In *Proceedings of the 24th Conference on Local Computer Networks* (1999), LCN'99, pp. 172–181.

[17] Figueiredo, D. R., Brandi, B., e de Souza e Silva, E. Mecanismos para Recuperação de Perdas de Pacotes de Voz na Internet. In *16th Brazilian Computer Networks Symposium 1998* (1998), pp. 52–67.

[18] Figueiredo, D. R., Pimentel, F., e de Souza e Silva, E. Implementação de um Aplicativo de Voz com Análise e Caracterização de seu Tráfego. In *XXIV Seminário Integrado de Software e Hardware 1997* (1997), pp. 531–540.

[19] Floyd, S., Jacobson, V., Liu, C.-G., McCanne, S., e Zhang, L. A Reliable Multicast Framework for Light-weight Sessions and Application Level

Framing. *IEEE/ACM Transactions on Networking 5*, 6 (dezembro de 1997), 784–803.

[20] Francis, P. Yoid: Extending the internet multicast architecture, abril de 2000. http://www.icir.org/yoid/docs/index.html.

[21] Grossglauser, M. Optimal deterministic timeouts for reliable scalable multicast. *IEEE Journal of Selected Areas in Communications 15*, 3 (1997), 422–433.

[22] Holbrook, H. W., Singhal, S. K., e Cheriton, D. R. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *SIGCOMM* (1995), pp. 328–341.

[23] Kurose, J. F., e Ross, K. W. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison Wesley, 2001.

[24] Lacher, M. S., Nonnenmacher, J., e Biersack, E. W. Performance Comparison of Centralized versus Distributed Error Recovery for Reliable Multicast. *IEEE/ACM Transactions on Networking 8*, 2 (2000), 224–224.

[25] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[26] Leão, R. M., de Souza e Silva, E., e de Lucena, S. C. A Set of tools for Traffic Modelling, Analysis and Experimentation. In *Proceedings of TOOLS* (2000).

[27] Levine, B. N., e Garcia-Luna-Aceves, J. J. A Comparison of Reliable Multicast Protocols. *Multimedia Systems 6*, 5 (1998), 334–348.

[28] Levine, B. N., e Garcia-Luna-Aceves, J. J. A Comparison of Known Classes of Reliable Multicast Protocols. *IEEE/ACM Transactions on Networking 8*, 2 (2000), 224–224.

[29] Lin, J. C., e Paul, S. Rmtp: A reliable multicast transport protocol. In *Proceedings of IEEE INFOCOM* (março de 1996), IEEE, pp. 1414–1425.

[30] Liu, C., Estrin, D., Shenker, S., e Zhang, L. Timer adjustment in SRM. *Technical report USC 97656, University of Southern California. July 1997.* (1997).

[31] Nonnenmacher, J., e Biersack, E. Optimal multicast feedback. In *INFOCOM (3)* (1998), pp. 964–971.

[32] Pendarakis, D., Shi, S., Verma, D., e marcel Waldvogel. ALMI: An Aplication Level Multicast Infrastructure. In *ACM USITS 2001* (San Francisco, USA, março de 2001), ACM.

[33] Ramakrishnan, S., e Jain, B. N. A Negative Acknowledgment with Periodic Polling Protocol for Multicast over Lan. In *IEEE Infocom* (1987), pp. 502–511.

[34] Scanferla, M., Azevedo, J. A., e Sadoc, D. The Reliable Multicast Library (RML) and Tangram II Whiteboard Developer Documentation.

[35] Strayer, W. T., Dempsey, B., e Weaver, A. *XTP: The Xpress Transfer Protocol.* Addison Wesley, 1992.

[36] Towsley, D. F., Kurose, J. F., e Pingali, S. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. *IEEE Journal of Selected Areas in Communications 15*, 3 (1997), 398–406.

[37] Tung, T. Mediaboard: A Shared Whiteboard Application for the MBone. Tese de Mestrado, University of California, Berkeley, 1997.

[38] Whetten, B., Montgomery, T., e Kaplan, S. M. A High Performance Totally Ordered Multicast Protocol. In *Dagstuhl Seminar on Distributed Systems* (1994), pp. 33–57.

[39] Yajnik, M., Kurose, J. F., e F.Towsley, D. Packet Loss Correlation in the MBone Multicast Network . *IEEE Global Internet Conference* (1996).

[40] Yamamoto, M., Kurose, J. F., Towsley, D. F., e Ikeda, H. A Delay Analysis of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. In *ACM SIGCOMM 1995* (1995), pp. 342–356.

[41] YAVATKAR, R., GRIFFOEN, J., E SUDAN, M. A Reliable Dissemination Protocol for Interactive Collaborative Applications. In *ACM Multimedia* (1995), pp. 333–344.

[42] ZHUNAG, S. Q., ZHAO, B. Y., JOSEPH, A. D., KATZ, R. H., E KUBIATOWICZ, J. D. Bayex: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *ACM NOSSDAV 2001* (New York, USA, junho de 2001), ACM, pp. 25–27.

Apêndice A

Código do Modelo Multicast Confiável

A.1 Eventos

A.1.1 Enviar_Dados

```
event= Enviar_Dados (UNI, MIN_DADOS, MAX_DADOS)
condition = (FALSE)
action = {

   int data_msg[3],
       sn_ultimo_seq,
       posicao,
       i,
       sn;

   int aux_cache[CACHE_SIZE],
       aux_pos_ultimo_seq,
       aux_pos_ultimo_id,
       aux_pos_inicial,
       aux_num_pacotes1,
       aux_dados_enviados,
       aux_sn_inicial;
```

```
get_st( aux_cache,"cache[]");
get_st( aux_dados_enviados,"dados_enviados");
get_st( &aux_pos_ultimo_seq,"pos_ultimo_seq");
get_st( &aux_pos_ultimo_id,"pos_ultimo_id");
get_st( &aux_pos_inicial,"pos_inicial");
get_st( &aux_sn_inicial,"sn_inicial");
get_st( &aux_num_pacotes1,"num_pacotes1");

/* enviar "num_pacotes1" pacotes */
for (i=0;i<aux_num_pacotes1;i++){

   /* achar ultimo sn em seq */
   if ( aux_pos_ultimo_seq >= aux_pos_inicial ){
      sn_ultimo_seq=aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial);
   }
   else{
      sn_ultimo_seq=(aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial))
                   + CACHE_SIZE;
   }

   sn=sn_ultimo_seq + 1;
   posicao=(sn % CACHE_SIZE);
   aux_cache[posicao]=DATA;

   data_msg[0]=DATA;
   data_msg[1]=ID;
   data_msg[2]=sn;
   msg(mcast_port,all,data_msg);
   aux_dados_enviados++;

   aux_pos_ultimo_seq=((aux_pos_ultimo_seq + 1) % CACHE_SIZE);
   aux_pos_ultimo_id=((aux_pos_ultimo_id + 1) % CACHE_SIZE);

   if ((aux_pos_ultimo_seq == aux_pos_inicial)||((sn == 0)
       && (aux_pos_inicial == -1))){

      aux_pos_inicial =((aux_pos_inicial + 1) % CACHE_SIZE);
      aux_sn_inicial++;
   }
}

set_st( "cache[]",aux_cache);
```

```
   set_st( "dados_enviados",aux_dados_enviados);
   set_st( "pos_ultimo_seq",aux_pos_ultimo_seq);
   set_st( "pos_ultimo_id",aux_pos_ultimo_id);
   set_st( "pos_inicial",aux_pos_inicial);
   set_st( "sn_inicial",aux_sn_inicial);

}:prob=prob_taxa1;

{

   int data_msg[3],
       sn_ultimo_seq,
       posicao,
       i,
       sn;

   int aux_cache[CACHE_SIZE],
       aux_pos_ultimo_seq,
       aux_pos_ultimo_id,
       aux_pos_inicial,
       aux_num_pacotes2,
       aux_dados_enviados,
       aux_sn_inicial;

   get_st( aux_cache,"cache[]");
   get_st( aux_dados_enviados,"dados_enviados");
   get_st( &aux_pos_ultimo_seq,"pos_ultimo_seq");
   get_st( &aux_pos_ultimo_id,"pos_ultimo_id");
   get_st( &aux_pos_inicial,"pos_inicial");
   get_st( &aux_sn_inicial,"sn_inicial");
   get_st( &aux_num_pacotes2,"num_pacotes2");

   /* enviar "num_pacotes2" pacotes */
   for (i=0;i<aux_num_pacotes2;i++){

      /* achar ultimo sn em seq */
      if ( aux_pos_ultimo_seq >= aux_pos_inicial ){
         sn_ultimo_seq=aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial);
      }
      else{
         sn_ultimo_seq=(aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial))
                       + CACHE_SIZE;
```

```
      }

      sn=sn_ultimo_seq + 1;
      posicao=(sn % CACHE_SIZE);
      aux_cache[posicao]=DATA;

      data_msg[0]=DATA;
      data_msg[1]=ID;
      data_msg[2]=sn;
      msg(mcast_port,all,data_msg);
      aux_dados_enviados++;

      aux_pos_ultimo_seq=((aux_pos_ultimo_seq + 1) % CACHE_SIZE);
      aux_pos_ultimo_id=((aux_pos_ultimo_id + 1) % CACHE_SIZE);

      if ((aux_pos_ultimo_seq == aux_pos_inicial) || ((sn == 0)
         && (aux_pos_inicial == -1))){

         aux_pos_inicial=((aux_pos_inicial + 1) % CACHE_SIZE);
         aux_sn_inicial++;
      }
   }

   set_st( "cache[]",aux_cache);
   set_st( "dados_enviados",aux_dados_enviados);
   set_st( "pos_ultimo_seq",aux_pos_ultimo_seq);
   set_st( "pos_ultimo_id",aux_pos_ultimo_id);
   set_st( "pos_inicial",aux_pos_inicial);
   set_st( "sn_inicial",aux_sn_inicial);


}:prob=prob_taxa2;

{

   int data_msg[3],
       sn_ultimo_seq,
       posicao,
       i,
       sn;

   int aux_cache[CACHE_SIZE],
```

```
    aux_pos_ultimo_seq,
    aux_pos_ultimo_id,
    aux_pos_inicial,
    aux_num_pacotes3,
    aux_dados_enviados,
    aux_sn_inicial;

get_st( aux_cache,"cache[]");
get_st( aux_dados_enviados,"dados_enviados");
get_st( &aux_pos_ultimo_seq,"pos_ultimo_seq");
get_st( &aux_pos_ultimo_id,"pos_ultimo_id");
get_st( &aux_pos_inicial,"pos_inicial");
get_st( &aux_sn_inicial,"sn_inicial");
get_st( &aux_num_pacotes3,"num_pacotes3");

/* enviar "num_pacotes3" pacotes */
for (i=0;i<aux_num_pacotes3;i++){

   /* achar ultimo sn em seq */
   if ( aux_pos_ultimo_seq >= aux_pos_inicial ){
      sn_ultimo_seq=aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial);
   }
   else{
      sn_ultimo_seq=(aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial))
                   + CACHE_SIZE;
   }

   sn=sn_ultimo_seq + 1;
   posicao=(sn % CACHE_SIZE);
   aux_cache[posicao]=DATA;

   data_msg[0]=DATA;
   data_msg[1]=ID;
   data_msg[2]=sn;
   msg(mcast_port,all,data_msg);
   aux_dados_enviados++;

   aux_pos_ultimo_seq=((aux_pos_ultimo_seq + 1) % CACHE_SIZE);
   aux_pos_ultimo_id=((aux_pos_ultimo_id + 1) % CACHE_SIZE);

   if ( (aux_pos_ultimo_seq == aux_pos_inicial) || ( (sn == 0)
      && (aux_pos_inicial == -1)) ){
```

```
            aux_pos_inicial=((aux_pos_inicial + 1) % CACHE_SIZE);
            aux_sn_inicial++;
        }
    }

    set_st( "cache[]",aux_cache);
    set_st( "dados_enviados",aux_dados_enviados);
    set_st( "pos_ultimo_seq",aux_pos_ultimo_seq);
    set_st( "pos_ultimo_id",aux_pos_ultimo_id);
    set_st( "pos_inicial",aux_pos_inicial);
    set_st( "sn_inicial",aux_sn_inicial);

}:prob=prob_taxa3;
```

A.1.2 Enviar_Nack

```
event= Enviar_Nack (UNI,A * D_NACK, (A + B) * D_NACK)
condition = (nack_cont > 0)
action = {

    int nack_msg[MSG_VEC_SIZE],
        sn_ultimo_seq,
        sn_ultimo_id,
        i,
        j,
        aux,
        status,
        posicao;

    int aux_cache[CACHE_SIZE],
        aux_pos_ultimo_seq,
        aux_pos_ultimo_id,
        aux_nack_cont,
        aux_ret_wait_cont,
        aux_pos_inicial,
        aux_sn_inicial;

    get_st( aux_cache,"cache[]");
    get_st( &aux_pos_ultimo_seq,"pos_ultimo_seq");
    get_st( &aux_pos_ultimo_id,"pos_ultimo_id");
```

```
get_st( &aux_nack_cont,"nack_cont");
get_st( &aux_ret_wait_cont,"ret_wait_cont");
get_st( &aux_pos_inicial,"pos_inicial");
get_st( &aux_sn_inicial,"sn_inicial");

/* achar ultimo sn em seq */
if ( aux_pos_ultimo_seq >= aux_pos_inicial ){
   sn_ultimo_seq=aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial);
}
else{
   sn_ultimo_seq=(aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial))
                 + CACHE_SIZE;
}
/* achar ultimo sn identificado */
if ( aux_pos_ultimo_id >= aux_pos_inicial ){
   sn_ultimo_id=aux_sn_inicial + (aux_pos_ultimo_id - aux_pos_inicial);
}
else{
   sn_ultimo_id=(aux_sn_inicial + (aux_pos_ultimo_id - aux_pos_inicial))
                + CACHE_SIZE;
}

/* Verificar se tem estado NACK na cache somente do ultimo em sequencia + 1 ateh
 * o ultimo identificado.*/

nack_msg[0]=NACK;
nack_msg[1]= ID;
j=2; /* começa a olhar os nacks a partir da terceira posição */
i=(sn_ultimo_seq + 1);

status = 0;
while ((i <= sn_ultimo_id) && (aux_nack_cont>0) && (status == 0)){
   posicao=(i % CACHE_SIZE);

   if (aux_cache[posicao] == NACK){
      if (j < (MSG_VEC_SIZE - 1)){
         nack_msg[j]=i;
         aux_cache[posicao]=RET_WAIT;
         set_cr(tempo_recup,i);
         aux=get_cr(nacks_enviados);
         aux++;
         set_cr(nacks_enviados,aux);
```

```
          aux_nack_cont--;
          aux_ret_wait_cont++;
          j++;
  }else
     status = 1;
      }
      i++;
   }

   nack_msg[j]=-1;
   msg(mcast_port,all,nack_msg);

   set_st( "cache[]",aux_cache);
   set_st( "nack_cont",aux_nack_cont);
   set_st( "ret_wait_cont",aux_ret_wait_cont);

};
```

A.1.3 Espera_Retransmissão

```
event= Espera_Retransmissao (UNI, A * D_WAIT, (A + B) * D_WAIT)
condition = (ret_wait_cont > 0)
action = {
   int sn_ultimo_seq,
       sn_ultimo_id,
       i,
       posicao;

   int aux_cache[CACHE_SIZE],
       aux_pos_ultimo_seq,
       aux_pos_ultimo_id,
       aux_nack_cont,
       aux_ret_wait_cont,
       aux_pos_inicial,
       aux_sn_inicial;

   get_st( aux_cache,"cache[]");
   get_st( &aux_pos_ultimo_seq,"pos_ultimo_seq");
   get_st( &aux_pos_ultimo_id, "pos_ultimo_id");
   get_st( &aux_nack_cont,"nack_cont");
   get_st( &aux_ret_wait_cont,"ret_wait_cont");
```

```
   get_st( &aux_pos_inicial,"pos_inicial");
   get_st( &aux_sn_inicial,"sn_inicial");

   /* achar ultimo sn em seq */
   if ( aux_pos_ultimo_seq >= aux_pos_inicial ){
      sn_ultimo_seq=aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial);
   }
   else{
      sn_ultimo_seq=(aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial))
                    + CACHE_SIZE;
   }
   /* achar ultimo sn identificado */
   if ( aux_pos_ultimo_id >= aux_pos_inicial ){
      sn_ultimo_id=aux_sn_inicial + (aux_pos_ultimo_id - aux_pos_inicial);
   }
   else{
      sn_ultimo_id=(aux_sn_inicial + (aux_pos_ultimo_id - aux_pos_inicial))
                   + CACHE_SIZE;
   }

    /* Verificar se tem estado RET_WAIT na cache somente do ultimo */
   /*  em sequencia + 1 até o ultimo identificado. */

   i=(sn_ultimo_seq + 1);

   while ((i <= sn_ultimo_id) && (aux_ret_wait_cont>0)){
      posicao=(i % CACHE_SIZE);

      if (aux_cache[posicao] == RET_WAIT){
         aux_cache[posicao]=NACK;
         aux_nack_cont++;
         aux_ret_wait_cont--;
      }
      i++;
   }

   set_st( "cache[]",aux_cache);
   set_st( "nack_cont",aux_nack_cont);
   set_st( "ret_wait_cont",aux_ret_wait_cont);

};
```

A.1.4 Enviar_Retransmissão

```
event= Enviar_Retransmissao (UNI, T_RET_MIN, T_RET_MAX)
condition = (ret_cont > 0)
action = {

   int ret_msg[3],
       sn_ultimo_seq,
       sn_ultimo_id,
       controle,
       i,
       aux,
       posicao;

   int aux_cache[CACHE_SIZE],
       aux_ret[RET_SIZE],
       aux_sn_inicial,
       aux_ret_cont,
       aux_pos_ret_inicial;

   get_st( aux_cache,"cache[]");
   get_st( aux_ret,"ret[]");
   get_st( &aux_sn_inicial,"sn_inicial");
   get_st( &aux_ret_cont,"ret_cont");
   get_st( &aux_pos_ret_inicial,"pos_ret_inicial");

   ret_msg[0]=RET_SEND;
   ret_msg[1]=ID;

   if (aux_pos_ret_inicial != -1){
      i = aux_ret[aux_pos_ret_inicial];
      aux_ret[aux_pos_ret_inicial]=-1;
      ret_msg[2]=i;
      aux_ret_cont--;

      /* só anda com o inicial se ainda tiver alguma retransmissão a ser enviada */
      if (aux_ret_cont != 0){
         aux_pos_ret_inicial =((aux_pos_ret_inicial + 1) % RET_SIZE);
      }

      if (i != -1) { /* se nao for uma retrans. suprimida */
         set_cr(sn_ret_enviada,i);
```

```
      aux=get_cr(ret_enviadas);
      aux++;
      set_cr(ret_enviadas,aux);
      msg(mcast_port,all,ret_msg);
    }
  }

  set_st( "cache[]",aux_cache);
  set_st( "ret[]",aux_ret);
  set_st( "sn_inicial",aux_sn_inicial);
  set_st( "ret_cont",aux_ret_cont);
  set_st( "pos_ret_inicial",aux_pos_ret_inicial);
};
```

A.1.5 Enviar_Refresh

```
event= Enviar_Refresh (DET, refresh_rate)
condition = (TRUE)
action = {

  int refresh_msg[3],
      aux_pos_ultimo_id,
      aux_pos_inicial,
      aux_sn_inicial,
      sn_ultimo_id;

  get_st( &aux_pos_ultimo_id,"pos_ultimo_id");
  get_st( &aux_pos_inicial,"pos_inicial");
  get_st( &aux_sn_inicial,"sn_inicial");

  /* achar ultimo sn identificado */
  if ( aux_pos_ultimo_id >= aux_pos_inicial ){
    sn_ultimo_id=aux_sn_inicial + (aux_pos_ultimo_id - aux_pos_inicial);
  }
  else{
    sn_ultimo_id=(aux_sn_inicial + (aux_pos_ultimo_id - aux_pos_inicial))
                 + CACHE_SIZE;
  }

  refresh_msg[0]=REFRESH;
  refresh_msg[1]=ID;
```

```
   refresh_msg[2]=sn_ultimo_id;

   msg(mcast_port,all,refresh_msg);
};
```

A.1.6 Tratar_Msg

```
event= Tratar_Msg(DET,tratar_msg_rate)
condition= ( msg_cont > 0 )
action= {

   int aux_msg_vec[MSG_VEC_SIZE],
      aux_cache[CACHE_SIZE],
      aux_ret[RET_SIZE],
      aux_pos_inicial,
      aux_sn_inicial,
      aux_pos_ultimo_seq,
      aux_pos_ultimo_id,
      aux_nack_cont,
      aux_ret_wait_cont,
      aux_ret_cont,
      aux_dados_recebidos,
      aux_msg_cont,
      aux_pos_ret_final,
      aux_pos_ret_inicial;

   int tipo,
       aux_tipo,
       aux_pos,
       aux,
       id,
       i,
       j,
       t,
       w,
       sn,
       aux_sn,
       aux_estado,
       posicao,
       aux_posicao,
       aux2_posicao,
```

```
    sn_ultimo_seq,
    sn_antigo,
    sn_ultimo_id,
    sn_aux,
    sn_limite,
    sn_refresh,
    sn_minimo,
    existe,
    pos;

 IntegerQueue fila_msg_aux( MSG_VEC_SIZE );

/* Inicializando variaveis locais */
get_st( &aux_dados_recebidos,"dados_recebidos");
get_st( &aux_msg_cont,"msg_cont");
get_st( aux_cache,"cache[]");
get_st( aux_ret,"ret[]");
get_st( &aux_pos_inicial,"pos_inicial");
get_st( &aux_sn_inicial,"sn_inicial");
get_st( &aux_pos_ultimo_seq,"pos_ultimo_seq");
get_st( &aux_pos_ultimo_id,"pos_ultimo_id");
get_st( &aux_nack_cont,"nack_cont");
get_st( &aux_ret_wait_cont,"ret_wait_cont");
get_st( &aux_ret_cont,"ret_cont");
get_st( &aux_pos_ret_final,"pos_ret_final");
get_st( &aux_pos_ret_inicial,"pos_ret_inicial");
get_st( fila_msg_aux, "fila_msg" );

restore_from_head( fila_msg_aux, aux_msg_vec );
tipo = aux_msg_vec[0];
aux_tipo=0;
existe=0;

/* achar ultimo sn em seq */
if ( aux_pos_ultimo_seq >= aux_pos_inicial ){
   sn_ultimo_seq=aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial);
}
else{
   sn_ultimo_seq=(aux_sn_inicial + (aux_pos_ultimo_seq - aux_pos_inicial))
                 + CACHE_SIZE;
}
```

```
/* achar ultimo sn identificado */
if ( aux_pos_ultimo_id >= aux_pos_inicial ){
   sn_ultimo_id=aux_sn_inicial + (aux_pos_ultimo_id - aux_pos_inicial);
}
else{
   sn_ultimo_id=(aux_sn_inicial + (aux_pos_ultimo_id - aux_pos_inicial))
                + CACHE_SIZE;
}
/* Fim das inicializacoes */

if ( tipo == NACK ){

   i=2; /* os sn's comecam da posicao 2 do vetor de msg */
   while (aux_msg_vec[i] != -1 ){
      existe=0;
      sn=aux_msg_vec[i];
      /* se o sn com o pedido de retransmissao ainda esta na cache */
      if ((sn >= sn_inicial) && (sn < sn_inicial + CACHE_SIZE)){
         posicao=( sn % CACHE_SIZE );
         aux_estado=aux_cache[posicao];

         if ( aux_estado == NACK ){
            aux_cache[posicao]= RET_WAIT;
            aux_nack_cont--;
            aux_ret_wait_cont++;
         }
         else if ( aux_estado == DATA ){

              /* não escalona a retransmissão */
              if (aux_pos_ret_inicial != -1)
                 if (aux_pos_ret_inicial <= aux_pos_ret_final)
                    for ( j=aux_pos_ret_inicial; j<=aux_pos_ret_final; j++)
                        if (aux_ret[j]==sn)
                           existe = 1;
                 else{
                    for ( j=aux_pos_ret_inicial; j<=(RET_SIZE - 1); j++)
                       if (aux_ret[j]==sn)
                          existe = 1;
                    if (existe !=1)
                       for ( j=0; j<=aux_pos_ret_final; j++)
                          if (aux_ret[j]==sn)
                             existe = 1;
```

```
              }

          if (existe !=1 ){ /* escalona a retransmissão */
             aux_pos = ((aux_pos_ret_final + 1) % RET_SIZE);
             /* verifica se cabe no vetor ret */
             if (aux_pos != aux_pos_ret_inicial){
               aux_pos_ret_final = ((aux_pos_ret_final + 1) % RET_SIZE);
               aux_ret[aux_pos_ret_final]=sn;

               aux=get_cr(escalona_vet_ret);
               aux++;
               set_cr(escalona_vet_ret,aux);

               /* anda com a posição inicial no começo e */
               /* quando aux_ret_cont deixar de ser = 0 */
               if ((aux_pos_ret_inicial == -1) || (aux_ret_cont == 0)){
                  aux_pos_ret_inicial = ((aux_pos_ret_inicial + 1) % RET_SIZE);
               }
               aux_ret_cont++;
               clone_ev("Enviar_Retransmissao");
               set_cr(escalona_nack_recebido,sn);
           }
           else{
               aux=get_cr(naoescalona_vet_ret);
               aux++;
               set_cr(naoescalona_vet_ret,aux);
              }
         }
      }
   }
   /* sn  a ser retransmitido nao encontra-se mais na cache */
   else{
     /* sn_minimo é o menor sn que pode ser retransmitido */
     /* em um determinado momento*/
     sn_minimo = sn_inicial - CACHE_VIRTUAL_SIZE;
     if (sn_minimo < 0){
        sn_minimo = 0;
     }
     if ((sn >= sn_minimo) && (sn < sn_inicial)){

          /* não escalona a retransmissão */
          if (aux_pos_ret_inicial != -1)
```

```
            if (aux_pos_ret_inicial <= aux_pos_ret_final)
               for ( j=aux_pos_ret_inicial; j<=aux_pos_ret_final; j++)
                  if (aux_ret[j]==sn)
                     existe = 1;
            else{
               for ( j=aux_pos_ret_inicial; j<=(RET_SIZE - 1); j++)
                  if (aux_ret[j]==sn)
                     existe = 1;
               if (existe !=1)
                  for ( j=0; j<=aux_pos_ret_final; j++)
                     if (aux_ret[j]==sn)
                        existe = 1;
             }

         if (existe !=1 ){ /* escalona a retransmissão */
          aux_pos = ((aux_pos_ret_final + 1) % RET_SIZE);
           /* verifica se cabe no vetor ret */
           if (aux_pos != aux_pos_ret_inicial){
              aux_pos_ret_final = ((aux_pos_ret_final + 1) % RET_SIZE);
              aux_ret[aux_pos_ret_final]=sn;

             /* anda com a posição inicial no começo e */
             /* quando aux_ret_cont deixar de ser = 0 */
             if ((aux_pos_ret_inicial == -1) || (aux_ret_cont == 0)){
                aux_pos_ret_inicial = ((aux_pos_ret_inicial + 1) % RET_SIZE);
              }
              aux_ret_cont++;
              clone_ev("Enviar_Retransmissao");
              set_cr(escalona_nack_recebido,sn);
           }

        }
    }
}

set_cr(sn_nack_recebido,sn);
id=aux_msg_vec[1];/* identificacao de quem enviou o nack.*/
switch ( id ){
    case 0: /* UFRJ */
        set_cr(sn_nack_recebido_UFRJ,sn);
        break;
    case 1: /* UFJF*/
```

```
                set_cr(sn_nack_recebido_UFJF,sn);
                break;
            case 3: /* UMASS */
                set_cr(sn_nack_recebido_UMASS,sn);
                break;
        }

        i++;
      }
    }
}

  else if ( tipo == REFRESH ){
      sn_refresh=aux_msg_vec[2];
      sn_aux = sn_ultimo_id +1;
      /* devem ser enviados NACKs para os sns ate o sn_limite */
      sn_limite = sn_ultimo_seq + CACHE_SIZE;

      if (sn_refresh < sn_limite)
         sn_limite=sn_refresh + 1;

      while (sn_aux < sn_limite){
         pos = (sn_aux % CACHE_SIZE);
         aux_cache[pos]= NACK;
         aux_nack_cont++;

         sn_aux++;
         aux_pos_ultimo_id = ((aux_pos_ultimo_id + 1) % CACHE_SIZE);

         /* se sn do ultimo identificado esta na mesma posicao do sn inicial */
         if (aux_pos_ultimo_id == aux_pos_inicial){
            aux_pos_inicial = ((aux_pos_inicial + 1) % CACHE_SIZE);
            aux_sn_inicial++;
         }
     }/*while*/
   }
   else {
     sn=aux_msg_vec[2];

      /* Verficando se o sn pode ser inserido na cache */
      if ((sn > sn_ultimo_seq) && (sn < sn_ultimo_seq + CACHE_SIZE)){
         posicao=( sn % CACHE_SIZE );
```

```
switch ( tipo ){
   case 1: /* DATA */
       set_cr(sn_dados_recebidos,sn);
       aux_tipo = 1;

   case 3: /* RET_SEND  Inserir dado na cache */

  if (tipo == 3){

            /* supressão de retransmissão */
            if (aux_pos_ret_inicial != -1)
              if (aux_pos_ret_inicial <= aux_pos_ret_final)
                 for ( j=aux_pos_ret_inicial; j<=aux_pos_ret_final; j++)
                    if (aux_ret[j]==sn){
                       aux_ret[j]=-1;
                       existe = 1;
                       aux=get_cr(retrans_suprimidas);
                       aux++;
                       set_cr(retrans_suprimidas,aux);
                    }
               else{
                  for ( j=aux_pos_ret_inicial; j<=(RET_SIZE - 1); j++)
                     if (aux_ret[j]==sn){
                       aux_ret[j]=-1;
                       existe = 1;
                       aux=get_cr(retrans_suprimidas);
                       aux++;
                       set_cr(retrans_suprimidas,aux);
                     }

                  if (existe !=1)
                    for ( j=0; j<=aux_pos_ret_final; j++)
                       if (aux_ret[j]==sn){
                          aux_ret[j]=-1;
                          existe = 1;
                          aux=get_cr(retrans_suprimidas);
                          aux++;
                          set_cr(retrans_suprimidas,aux);
                       }
                    }
               }
```

```
      if ((existe != 1) || (tipo == 1)){
         if ((sn == 0) && (aux_pos_inicial == -1)){
            /* Recebi o primeiro dado da cache */
            aux_pos_inicial = ((aux_pos_inicial + 1) % CACHE_SIZE);
            aux_sn_inicial++;
          }

  if ( sn == (sn_ultimo_seq + 1) ){
  /* sn em sequencia */
    aux_pos_ultimo_seq = ((aux_pos_ultimo_seq + 1) % CACHE_SIZE);
    /* posicao do sn posterior ao sn recebido */
    aux_sn = sn + 1;
    aux2_posicao= ((aux_sn) % CACHE_SIZE);

    /* enquanto tiver um dado na posicao seguinte e o sn a
    ser verificado for menor que o ultimo identificado */
    while ((aux_cache[aux2_posicao] == DATA) &&
                 (aux_sn <= sn_ultimo_id)) {
      aux_pos_ultimo_seq = ((aux_pos_ultimo_seq + 1) % CACHE_SIZE);
      aux_sn++;
      aux2_posicao= (aux_sn % CACHE_SIZE);
    }

    if ( (aux_pos_ultimo_seq == aux_pos_inicial) && ( sn != 0 ) ){
              aux_pos_inicial = ((aux_pos_inicial + 1) % CACHE_SIZE);
              aux_sn_inicial++;
    }

    /* supressão de nack */
    if (aux_cache[posicao] == NACK){
              aux_nack_cont--;
              aux=get_cr(nacks_suprimidos);
              aux++;
              set_cr(nacks_suprimidos,aux);
    }
    if (aux_cache[posicao] == RET_WAIT){
              aux_ret_wait_cont--;
              aux=get_cr(nacks_suprimidos);
              aux++;
              set_cr(nacks_suprimidos,aux);
         }
```

```
        aux_cache[posicao]=DATA;
        aux_dados_recebidos++;
}
else {
/* sn fora de sequencia */
/* Verficar se esta na regiao critica:
* Pertencem a regiao critica todos os sn maiores que o sn inicial
* e menores que o ultimo sn em sequencia
*/
    if (posicao >= aux_pos_inicial) {
        sn_antigo = sn_inicial + (posicao - aux_pos_inicial);
    }
    else{
        sn_antigo = (sn_inicial + (posicao - aux_pos_inicial))
                    + CACHE_SIZE;
    }

    if (sn_antigo < sn_ultimo_seq){
    /* Esta na regiao critica.
    * Zerar as posicoes do inicio da cache ateh a posicao
    * que este vai sn vai ser inserido
    */

        while ((aux_sn_inicial + CACHE_SIZE) <= sn){

            aux_cache[aux_pos_inicial]=0;
            aux_pos_inicial = ((aux_pos_inicial + 1) % CACHE_SIZE);
            aux_sn_inicial++;
        }
    }
    else{ /* Estah fora da regiao critica */
        /* supressão de nack */
        if (aux_cache[posicao] == NACK){
                  aux_nack_cont--;
                  aux=get_cr(nacks_suprimidos);
                  aux++;
                  set_cr(nacks_suprimidos,aux);
        }
        else if (aux_cache[posicao] == RET_WAIT){
                  aux_ret_wait_cont--;
                  aux=get_cr(nacks_suprimidos);
                  aux++;
```

```
                     set_cr(nacks_suprimidos,aux);
                  }
               }

               if ((sn != sn_antigo)||((sn == sn_antigo) &&
                     (aux_cache[posicao] != DATA))){
                  aux_dados_recebidos++;
               }
               aux_cache[posicao]=DATA;
            }

            if ( sn > sn_ultimo_id ){
            /* Enviar nack do ultimo identificado ate esse sn */
               for (i=(sn_ultimo_id + 1);i<sn;i++){

                  aux_posicao=(i % CACHE_SIZE);

                  if (aux_cache[aux_posicao] == 0){
                     aux_cache[aux_posicao]=NACK;
                     aux_nack_cont++;
                  }
               }

               /* Atualizando a posicao do ultimo identificado */
               aux_pos_ultimo_id = posicao;
               if ((aux_pos_ultimo_id == aux_pos_inicial) && ( sn != 0 )){
                   aux_pos_inicial = ((aux_pos_inicial + 1) % CACHE_SIZE);
                   aux_sn_inicial++;
               }
            }
                if (aux_tipo == 0){
                   t=sn+ 10000;
                   set_cr(tempo_recup,t);
             }
                  }
           break;

      } /* switch */
   } /* pacote pode ser inserido na cache */
}
aux_msg_cont--;
```

```
   /* Atualizando as variaveis de estado */
   set_st( "dados_recebidos",aux_dados_recebidos);
   set_st( "msg_cont",aux_msg_cont);
   set_st( "cache[]",aux_cache);
   set_st( "ret[]",aux_ret);
   set_st( "pos_inicial",aux_pos_inicial);
   set_st( "sn_inicial",aux_sn_inicial);
   set_st( "pos_ultimo_seq",aux_pos_ultimo_seq );
   set_st( "pos_ultimo_id",aux_pos_ultimo_id );
   set_st( "nack_cont",aux_nack_cont);
   set_st( "ret_wait_cont",aux_ret_wait_cont);
   set_st( "ret_cont",aux_ret_cont);
   set_st( "pos_ret_final",aux_pos_ret_final);
   set_st( "pos_ret_inicial",aux_pos_ret_inicial);
   set_st( "fila_msg",fila_msg_aux);

};
```

A.1.7 Delay01

```
event= Delay01 (GAUSS, DELAY01, VARIANCIA01)
condition= (Fila01_Saida > 0)
action= {

    IntegerQueue fila01(MSG_SIZE);

    int fila01_saida,
        aux_msg[MSG_SIZE];

    get_st(fila01,"Fila01");
    get_st(fila01_saida,"Fila01_Saida");

    restore_from_head(fila01,aux_msg);
    msg(PORTA02,all,aux_msg);
    fila01_saida--;

    set_st("Fila01",fila01);
    set_st("Fila01_Saida",fila01_saida);

};
```

A.1.8 Delay02

```
event= Delay02 (GAUSS, DELAY02, VARIANCIA02)
condition= (Fila02_Entrada > 0)
action= {
    IntegerQueue fila02(MSG_SIZE);

    int fila02_entrada,
        aux_msg[MSG_SIZE];

    get_st(fila02,"Fila02");
    get_st(fila02_entrada,"Fila02_Entrada");

    restore_from_head(fila02,aux_msg);
    msg(PORTA01,all,aux_msg);
    fila02_entrada--;

    set_st("Fila02", fila02);
    set_st("Fila02_Entrada",fila02_entrada);

};
```

A.2 Mensagens

A.2.1 Objetos: UFRJ, UFF, UMASS, UFJF

```
msg_rec=mcast_port
action= {

   int aux_msg_cont,
       aux_fila_msg[MSG_VEC_SIZE];

   float aux_T_RET_MIN, aux_T_RET_MAX;

   IntegerQueue fila_msg_aux(MSG_VEC_SIZE);

   get_st( &aux_T_RET_MIN,"T_RET_MIN");
   get_st( &aux_T_RET_MAX,"T_RET_MAX");
   get_st( fila_msg_aux,"fila_msg");
   get_st( &aux_msg_cont,"msg_cont");
```

```
   get_msg_data(aux_fila_msg);

   /* Coloca a mensagem recebida na fila para simular delay do canal */
   save_at_tail( fila_msg_aux, aux_fila_msg );

   /* Clonar evento de tratamento de mensagem */
   clone_ev("Tratar_Msg");

   aux_msg_cont++;

   if(aux_fila_msg[0]==NACK){
      switch(aux_fila_msg[1]){
         case 1:
            aux_T_RET_MIN=E * D_RET_1;
            aux_T_RET_MAX=(E + F) * D_RET_1;
            break;

         case 2:
            aux_T_RET_MIN=E * D_RET_2;
            aux_T_RET_MAX=(E + F) * D_RET_2;
            break;

         case 3:
            aux_T_RET_MIN=E * D_RET_3;
            aux_T_RET_MAX=(E + F) * D_RET_3;
            break;
      }
   }
   set_st( "T_RET_MIN",aux_T_RET_MIN);
   set_st( "T_RET_MAX",aux_T_RET_MAX);
   set_st( "msg_cont",aux_msg_cont);
   set_st( "fila_msg",fila_msg_aux);
};
```

A.2.2 Objetos: rede01, rede02, rede03, rede04

```
msg_rec=PORTA01
action={

    IntegerQueue fila01(MSG_SIZE);
```

```
    int fila01_saida,
        aux_msg[MSG_SIZE];

    get_st(fila01,"Fila01");
    get_st(fila01_saida,"Fila01_Saida");

    get_msg_data(aux_msg);

    save_at_tail(fila01,aux_msg);
    fila01_saida++;
    clone_ev("Delay01");

    set_st("Fila01",fila01 );
    set_st("Fila01_Saida",fila01_saida);

}:prob=1-PROB_PERDA01;
{ /* Simulação de perda de dados e retransmissões */

    IntegerQueue fila01(MSG_SIZE);

    int fila01_saida,
        aux_msg[MSG_SIZE];

    get_st(fila01,"Fila01" );
    get_st(fila01_saida,"Fila01_Saida") ;

    get_msg_data(aux_msg);

    if((aux_msg[0]==NACK) || (aux_msg[0]==REFRESH)){

       save_at_tail(fila01,aux_msg);
       fila01_saida++;
       clone_ev("Delay01");

    }

    set_st("Fila01",fila01);
    set_st("Fila01_Saida",fila01_saida);

}:prob=PROB_PERDA01;
```

```
msg_rec=PORTA02
action={
    IntegerQueue fila02(MSG_SIZE);

    int fila02_entrada,
        aux_msg[MSG_SIZE];

    get_st(fila02,"Fila02");
    get_st(fila02_entrada,"Fila02_Entrada");

    get_msg_data(aux_msg);

    save_at_tail(fila02,aux_msg);
    fila02_entrada++;
    clone_ev("Delay02");

    set_st("Fila02",fila02);
    set_st("Fila02_Entrada",fila02_entrada);

}:prob=1-PROB_PERDA02;
{ /* Simulação de perda de dados e retransmissões */

    IntegerQueue fila02(MSG_SIZE);

    int fila02_entrada,
        aux_msg[MSG_SIZE];

    get_st(fila02,"Fila02");
    get_st(fila02_entrada,"Fila02_Entrada");

    get_msg_data(aux_msg);

    if((aux_msg[0]==NACK) || (aux_msg[0]==REFRESH)){

       save_at_tail(fila02,aux_msg);
       fila02_entrada++;
       clone_ev("Delay02");

    }

    set_st("Fila02",fila02);
    set_st("Fila02_Entrada",fila02_entrada);
```

```
}:prob=PROB_PERDA02;
```

A.2.3 Objeto: *spliter*

```
msg_rec=PORTA01
action={

   int aux_msg[MSG_VEC_SIZE];

   get_msg_data(aux_msg);

   /* Enviando a mensagem para as outras portas */
   msg(PORTA02,all,aux_msg);
   msg(PORTA03,all,aux_msg);
   msg(PORTA04,all,aux_msg);

};
msg_rec=PORTA02
action={

   int aux_msg[ MSG_VEC_SIZE ];

   get_msg_data(aux_msg);

   /* Enviando a mensagem para as outras portas */
   msg( PORTA01,all,aux_msg );
   msg( PORTA03,all,aux_msg );
   msg( PORTA04,all,aux_msg );
};
msg_rec=PORTA03
action={
   int aux_msg[MSG_VEC_SIZE];

   get_msg_data(aux_msg);

   /* Enviando a mensagem para as outras portas */
   msg(PORTA01,all,aux_msg);
   msg(PORTA02,all,aux_msg);
   msg(PORTA04,all,aux_msg);
};
```

```
msg_rec=PORTA04
action={

   int aux_msg[MSG_VEC_SIZE];

   get_msg_data(aux_msg);

   /* Enviando a mensagem para as outras portas */
   msg(PORTA01,all,aux_msg);
   msg(PORTA02,all,aux_msg);
   msg(PORTA03,all,aux_msg);
};
```

A.2.4 Objeto: *proxy*

```
msg_rec=PORTA01
action={

   int aux_msg[MSG_VEC_SIZE];

   get_msg_data(aux_msg);

   /* Enviando a mensagem para as outras portas */
   msg(PORTA02,all,aux_msg);
   msg(PORTA03,all,aux_msg);
   msg(PORTA04,all,aux_msg);
};

msg_rec=PORTA02
action={

   int aux_msg[ MSG_VEC_SIZE ];

   get_msg_data(aux_msg);

   /* Enviando a mensagem para a porta 01 */
   msg( PORTA01,all,aux_msg );
};

msg_rec=PORTA03
action={
```

```
   int aux_msg[MSG_VEC_SIZE];

   get_msg_data(aux_msg);

   /* Enviando a mensagem para a porta 01 */
   msg(PORTA01,all,aux_msg);
};

msg_rec=PORTA04
action={

   int aux_msg[MSG_VEC_SIZE];

   get_msg_data(aux_msg);

  /* Enviando a mensagem para a porta 01 */
   msg(PORTA01,all,aux_msg);
};
```

A.3 Declarações

A.3.1 Objetos: UFRJ, UFF, UMASS, UFJF

```
Const
    Integer: CACHE_SIZE, /* tamanho da cache */
             RET_SIZE, /* tamanho do vetor de retransmissões */
             MAX_DATA, /* número máximo de dados enviados */
             CACHE_VIRTUAL_SIZE, /* tamanho da cache virtual (retransmissão) */
             MSG_VEC_SIZE, /* tamanho da mensagem */
             DATA, /* na cache indica dado recebido, na mensagem indica tipo dado */
             NACK, /* na cache indica enviar NACK, na mensagem indica tipo nack */
             RET_WAIT, /* na cache indica espera ret., nao existe msg deste tipo */
             REFRESH, /* nao tem significado na cache, na msg indica tipo refresh */
             RET_SEND, /* nao tem significado na cache, na msg indica tipo retrans. */
             ID, /* identificação do membro do grupo multicast */
             MIN_DADOS,
             MAX_DADOS, /* tempo min. e max. do intervalo entre o envio das msgs */
             A,B,C,D,E,F; /* constantes usadas no cálculo dos temporizadores */

    Float:   D_NACK, /* atraso considerado no cálculo do temporizador T_NACK */
```

```
          D_WAIT, /* atraso considerado no cálculo do temporizador T_WAIT */
          D_RET_1, /* atraso considerado no cálculo do temporizador T_RET  */
          D_RET_2, /* atraso considerado no cálculo do temporizador T_RET  */
          D_RET_3, /* atraso considerado no cálculo do temporizador T_RET  */
          refresh_rate, /* taxa do evento enviar_refresh */
          tratar_msg_rate,/* taxa do evento tratar_msg */
          prob_taxa1,prob_taxa2,
          prob_taxa3; /* prob. de enviar dados com uma determinada taxa */

   Port: mcast_port; /* porta de comunicação do evento */

State Var

   Integer: cache[CACHE_SIZE], /* vetor que representa a cache */
          ret[RET_SIZE], /* vetor com as retransmissões a serem enviadas */
          pos_inicial, /* posição inicial da cache */
          sn_inicial, /* sn inicial da cache */
          nack_cont, /* indica quantos NACKs estao para serem enviados */
          ret_cont, /* indica quantas retransmissoes estão para serem enviadas */
          ret_wait_cont, /* indica quantas retransmissoes estao sendo esperadas */
          msg_cont, /* indica quantas msgs estão na fila para serem tratadas */
          pos_ultimo_seq, /* indica a posição do último pacote recebido em seq.*/
          pos_ultimo_id, /* indica a posição do último pacote identificado */
          dados_enviados, /* número de dados enviados pelo emissor */
          dados_recebidos, /* número de dados recebidos pelo receptor */
          pos_ret_inicial, /* pos. do próximo pacote a ser retrans. no vetor ret */
          pos_ret_final, /* pos. do último pacote a ser retrans. no vetor de ret.*/
          num_pacotes1,num_pacotes2,
          num_pacotes3; /* número de pacotes a serem enviados */

   IntegerQueue: fila_msg(MSG_VEC_SIZE); /* fila com as mensagens recebidas */

   Float: T_RET_MIN, /* tempo min. do intervalo entre envio das ret. solicitadas */
        T_RET_MAX; /* tempo max. do intervalo entre envio das ret. solicitadas */
```

A.3.2 Objetos: rede01, rede02, rede03, rede04

```
Const
   Float: DELAY01, /* delay aplicado aos pacotes procedentes da PORTA01 */
        DELAY02, /* delay aplicado aos pacotes procedentes da PORTA02 */
        VARIANCIA01, /* variancia para o DELAY01 */
```

```
          VARIANCIA02, /* variancia para o DELAY01 */
          PROB_PERDA01, /* prob. de perda dos pacotes procedentes da PORTA01 */
          PROB_PERDA02, /* prob. de perda dos pacotes procedentes da PORTA01 */
          NACK, /* indica que a mensagem é do tipo nack */
          REFRESH; /* indica que a mensagem é do tipo refresh */

   Integer: MSG_SIZE; /* tamanho maximo das mensagens recebidas */

   Port: PORTA01,
         PORTA02;

State Var
   IntegerQueue: Fila01(MSG_SIZE), /* armazena os pcts que serao enviados pela PORTA2 */
                 Fila02(MSG_SIZE); /* armazena os pcts que serao enviados pela PORTA1 */

   Integer: Fila01_Saida, /* quantidade de pacotes na Fila01 */
            Fila02_Entrada; /* quantidade de pacotes na Fila01 */
```

A.3.3 Objeto: *spliter* e *proxy*

```
Const
  Integer: MSG_VEC_SIZE; /* tamanho do vetor de mensagem */

  Port: PORTA01, PORTA02, PORTA03, PORTA04;
```

A.4 Recompensas

```
impulse_reward = pacotes_nacks_enviados
   event = Enviar_Nack,1
   value = 1;

impulse_reward = ret_enviadas
   event = Enviar_Retransmissao,1
   value = 1;

rate_reward = dados_env
   condition = (FALSE)
   value = 0;
```

```
rate_reward = tempo_recup
   condition = (FALSE)
   value = 0;

rate_reward = retrans_suprimidas
   condition = (FALSE)
   value = 0;

rate_reward = nacks_suprimidos
   condition = (FALSE)
   value = 0;

rate_reward = escalona_nack_recebido
   condition = (FALSE)
   value = 0;

rate_reward = escalona_vet_ret
   condition = (FALSE)
   value = 0;

rate_reward = naoescalona_vet_ret
   condition = (FALSE)
   value = 0;

rate_reward = sn_ret_enviada
   condition = (FALSE)
   value = 0;

rate_reward = sn_dados_recebidos
   condition = (FALSE)
   value = 0;

rate_reward = sn_ret_recebida
   condition = (FALSE)
   value = 0;

rate_reward = sn_nack_recebido
   condition = (FALSE)
   value = 0;

rate_reward = sn_ret_recebida_UFF
   condition = (FALSE)
```

```
   value = 0;

rate_reward = sn_ret_recebida_UFRJ
   condition = (FALSE)
   value = 0;

rate_reward = sn_ret_recebida_UMASS
   condition = (FALSE)
   value = 0;

rate_reward = sn_ret_recebida_UFJF
   condition = (FALSE)
   value = 0;

rate_reward = sn_nack_recebido_UFF
   condition = (FALSE)
   value = 0;

rate_reward = sn_nack_recebido_UMASS
   condition = (FALSE)
   value = 0;

rate_reward = sn_nack_recebido_UFJF
   condition = (FALSE)
   value = 0;

rate_reward = sn_nack_recebido_UFRJ
   condition = (FALSE)
   value = 0;
```