UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ELETRÔNICA E DE COMPUTAÇÃO

Estudo, Implementação e Análise de Métricas Baseadas na Qualidade do Enlace para o Protocolo OLSR

Autor:

Felipe Ortigão Sampaio Buarque Schiller

Orientador:

Prof. Luís Henrique Maciel Kosmalski Costa, Dr.

Examinador:

Prof. Marcelo Gonçalves Rubinstein, D. Sc.

Examinador:

Prof. José Gabriel Rodríguez Carneiro Gomes, Ph.D.

Examinador:

Miguel Elias Mitre Campista, M.Sc.

Examinador:

Igor Monteiro Moraes, M.Sc.

DEL
Janeiro de 2007

Agradecimentos

Cumpre-me agradecer a todos aqueles que direta e indiretamente contribuíram para que pudesse levar a bom termo a elaboração deste projeto final.

Primeiramente agradeço a meus pais, pelo exemplo de vida e dedicação aos estudos que me ofertaram, além de todo o apoio para a realização desse projeto; aos meus irmãos pela paciência nas horas que eu estava indisponível para ajudar com outros assuntos; à minha avó por todas as preces e apoio; ao meu orientador Luís Henrique, que com todas as críticas e conselhos aperfeiçoou o meu texto; aos meus colegas de graduação e do GTA por todo o convívio e ajuda prestados; a todos os professores do Departamento de Eletrônica e Computação por todo o aprendizado adquirido nesses cinco anos; enfim a todos os meus parentes e amigos que suportaram minha ausência.

Resumo

Para a comunicação com múltiplos saltos em redes ad hoc sem fio é necessária a utilização de protocolos de roteamento. Um dos protocolos mais difundidos, foco desse trabalho, é o OLSR (*Optimized Link State Routing Protocol*), sendo inclusive empregado nas denominadas redes em malha sem fio. Nas implementações existentes do OLSR são utilizadas duas métricas, número de saltos, onde a menor distância entre dois nós é a única levada em conta para determinação da rota, e a ETX (*Expected Transmission Count*), que utiliza a taxa de perda de pacotes para determinar o melhor caminho.

Entretanto, nenhuma dessas métricas leva em conta o tempo de transmissão entre dois nós, cuja solução é a proposta da métrica ETT (*Estimated Transmission Time*). Assim, foi desenvolvido um *plugin* para implementar a métrica ETT no protocolo OLSR, sendo realizados diversos testes para avaliar seu desempenho. A grande vantagem dessa escolha é de não haver necessidade de modificar o código, privilegiando desta forma a portabilidade.

A plataforma escolhida como parte do ambiente de testes, além de micro-computadores, foi o roteador Linksys WRT54g, que por ser uma plataforma dedicada ao meio sem fio e de fácil posicionamento, devido ao seu tamanho, é ideal para a realização dos testes. Outra vantagem sua é que também é possível rodar o sistema operacional aberto Linux.

Palavras-chave

Redes sem fio

Redes ad hoc

Protocolo de roteamento

Métrica de roteamento

Redes em malha

Lista de Acrônimos

ACK	*Acknowledgement;*
ANSN	*Advertised Neighbor Sequence Number;*
DSDV	*Destination-Sequenced Distance Vector;*
DSR	*Dynamic Source Routing Protocol;*
ETT	*Estimated Transmission Time;*
ETX	*Expected Transmission Count;*
GNU	*Acrônimo de GNU is Not UNIX;*
GNU GPL	*GNU General Public License;*
HNA	*Host and Network Association;*
IANA	*Internet Assigned Numbers Authority;*
IP	*Internet Protocol;*
IPC	*Interprocess Communication;*
JFFS2	*Journaling Flash File System version 2;*
LQ	*Link Quality;*
MID	*Multiple Interface Declaration;*
MIPS	*Millions of Instructions Per Second;*
MPR	*Multipoint Relay;*
NLQ	*Neighbor Link Quality;*
OLSR	*Optimized Link State Routing Protocol;*
RFC	*Request For Comments;*
RIP	*Routing Information Protocol;*
RISC	*Reduced Instruction Set Computer;*
SDK	*Software Development Kit;*
TTL	*Time To Live;*
UDP	*User Datagram Protocol;*
ZRP	*Zone Routing Protocol;*

Sumário

Resumo iii

Palavras-chave iv

Lista de Acrônimos v

Lista de Figuras viii

Lista de Tabelas ix

1. Introdução 1

2. Protocolos de Roteamento 5

2.1. DSR 5

2.1.1. Descoberta de Rotas 6

2.1.2. Manutenção de Rotas 7

2.2. AODV 8

2.2.1. Descoberta de Rotas 9

2.2.2. Manutenção de Rotas 10

2.3. DSDV 10

2.4. ZRP 11

2.5. OLSR 11

2.5.1. Formato do Pacote 13

2.5.2. Mensagem de HELLO 15

2.5.3. Cálculo do MPR 17

2.5.4. Descoberta de Topologia 19

2.5.5. Cálculo da Tabela de Roteamento 20

3. Métricas de Roteamento 23

3.1. ETX 23

3.2. ETT 26
4. Implementação da Métrica ETT no OLSR 29
4.1. A Plataforma 29
4.1.1. Compilando para a Plataforma 30
4.2. Código Fonte 31
4.3. Cenário de Teste 36
4.4. Resultados 40
5. Conclusão 47
6. Referências 49
7. Apêndice 51
7.1. Código Fonte do Plugin 51
7.1.1. olsrd_plugin.c 51
7.1.2. olsrd_ett.h 52
7.1.2. olsrd_ett.c 55
7.1.3. Olsrd_ett_rot.conf 70
7.2. Código dos arquivos necessários para gerar o IPK 76
7.2.1. Makefile 76
7.2.2. Config.in 77
7.2.3. ettmetric.control 77
7.3. Scripts de teste 77
7.3.1. roda_tamanho_pacote.sh 77
7.3.2 ic_miguel.awk 79
7.3.3 analise_ping_perda.awk 83
7.3.4 analise_ping_troca.awk 83

Lista de Figuras

Figura 1: Rede Infra-estruturada........ 1

Figura 2: Rede Ad hoc........ 1

Figura 3: Rede Metropolitana - Adaptado de [7]........ 2

Figura 4: Descoberta de rotas no DSR........ 6

Figura 5: Inundação normal........ 12

Figura 6: Inundação com MPR........ 12

Figura 7: Formato do pacote do OLSR........ 14

Figura 8: Formato da mensagem de HELLO........ 15

Figura 9: Estado do enlace........ 16

Figura 10: Mensagem de controle de topologia........ 19

Figura 11: Mensagem LQ HELLO........ 25

Figura 12: Mensagens LQ TC........ 26

Figura 13: Linksys WRT54g........ 29

Figura 14: Topologia da métrica contagem de saltos........ 41

Figura 15: Topologia da métrica ETX........ 42

Figura 16: Topologia da métrica ETT........ 43

Figura 17: Perda de pacotes para a configuração padrão........ 44

Figura 18: Troca de rota para a configuração padrão........ 44

Figura 19: Perda de pacotes para a configuração não-padrão........ 45

Figura 20: Troca de rota para a configuração não-padrão........ 45

Lista de Tabelas

Tabela 1: Formato da tabela de roteamento..20

Tabela 2: Configurações dos roteadores do ambiente de testes. 30

Tabela 3: Resumo dos arquivos de configuração. 40

1. Introdução

O padrão IEEE 802.11, que define o funcionamento das redes sem fio locais, é atualmente um sucesso comercial. O seu baixo custo de implantação e manutenção o torna presente em redes domiciliares, centros comerciais, aeroportos e até restaurantes. Já existem até centros urbanos onde foram instaladas redes com o objetivo de cobrir toda uma cidade, como por exemplo, em Taipei, em Taiwan [17] e São Francisco, nos Estados Unidos [18].

Existem dois tipos de redes 802.11, as infra-estruturadas e as ad hoc. As redes infra-estruturadas são formadas por um ponto de acesso onde todas as estações se conectam, sendo possível através dele o acesso tanto a outros nós da rede como à Internet. Já nas redes ad hoc, já não existe uma unidade centralizadora, os nós se comunicam diretamente uns com os outros. Nós fora de alcance mútuo utilizam o encaminhamento por múltiplos saltos para se comunicarem, o que exige um protocolo de roteamento e a colaboração dos nós da rede. As Figuras 1 e 2 ilustram a diferença entre essas redes, sendo importante notar que no caso de redes ad hoc, sem o uso de um protocolo de roteamento, só há comunicação entre nós que se alcançam.

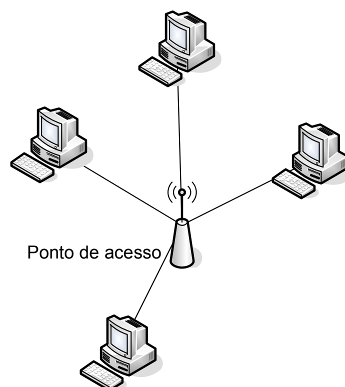


Figura 1: Rede Infra-estruturada.

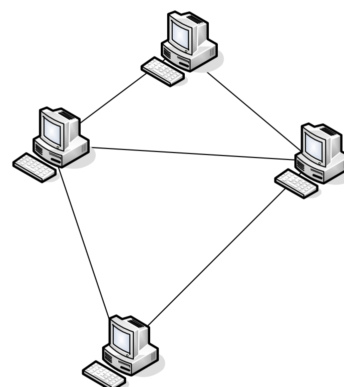


Figura 2: Rede Ad hoc.

Os protocolos de roteamento para redes ad hoc têm como meta que nós que não estejam diretamente ao alcance um do outro, possam se comunicar através de outros nós. Entretanto, a escolha do melhor caminho vai depender da métrica utilizada pelo protocolo, como por exemplo, o caminho com o menor número de saltos ou a menor taxa de erro.

As redes em malha sem fio são uma evolução das redes ad hoc, visando estender a Internet em locais de difícil acesso a um baixo custo. O objetivo das redes em malha é prover acesso à rede em qualquer lugar. A diferença para as redes ad hoc é que o funcionamento delas é baseado em um *backbone* sem fio, fixo, onde não é necessária a preocupação com consumo de bateria nem mobilidade. Desta forma, os protocolos podem se dedicar a busca pela melhor rota através da qualidade dos enlaces, que é um fator muito variável nas redes sem fio. Isso é devido a constantes mudanças do meio, como interferência de outros equipamentos, o deslocamento de indivíduos e o deslocamento dos próprios nós.

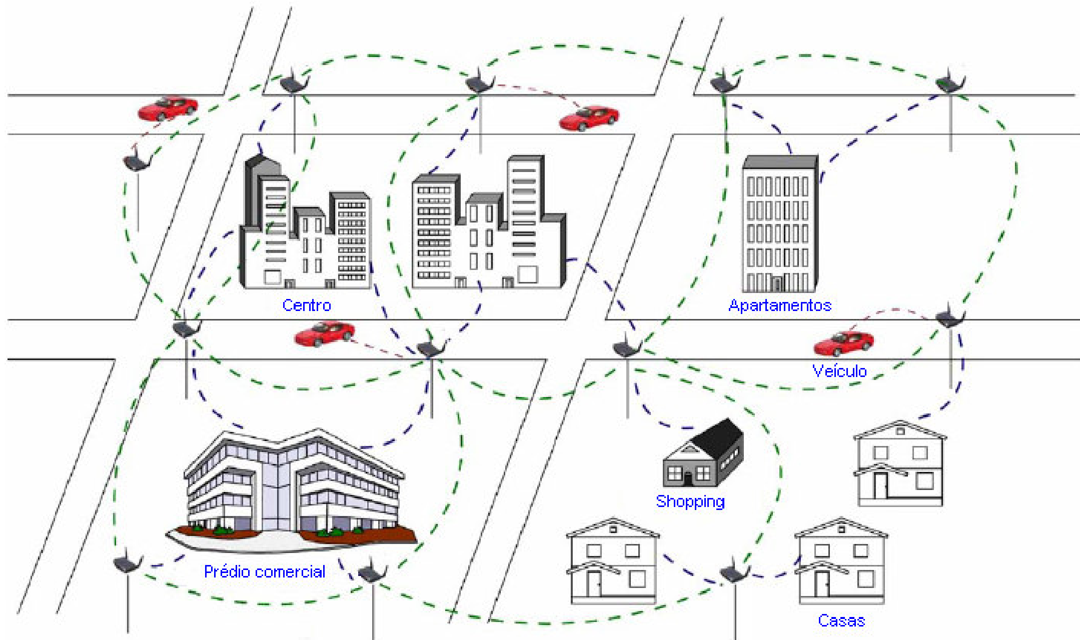


Figura 3: Rede Metropolitana - Adaptado de [7].

Os cenários de aplicação dessas redes são inúmeros, como em redes domiciliares, acessos comunitários, redes empresariais, redes metropolitanas, sistemas de transporte e

sistemas de segurança. A Figura 3 ilustra como ela poderia ser aplicada em redes metropolitanas.

Observe na Figura 3 que há uma intercomunicação entre os diversos componentes de uma cidade, como residências, prédios comerciais, veículos e enfim, qualquer dispositivo que possa se conectar a rede nesse meio.

Este trabalho se concentra nas métricas de roteamento, através do estudo do protocolo *Optimized Link State Routing* (OLSR), utilizado para redes em malha. A sua escolha foi feita por ser um protocolo bastante testado e utilizado, além de possuir código aberto.

Dentre as implementações do OLSR existentes, esse projeto está baseado na desenvolvida por Andreas Tønnesen em sua tese [3] e que permite a inclusão de *plugins*. Esta versão vem sendo mantida e modificada, inclusive tendo sido adicionada a métrica ETX (*Expected Transmission Count*) e é utilizada em grandes projetos, principalmente o Freifunk [4] na Alemanha, onde várias cidades construíram suas próprias redes. Todas as redes são abertas e colaborativas, sendo a maior delas em Berlim, com mais de 500 nós.

A parte prática deste trabalho utiliza, além de microcomputadores, roteadores da Cisco, Linksys WRT54g. Por rodarem o sistema operacional aberto Linux e o protocolo de roteamento OLSR, inclusive sendo utilizado com sucesso no projeto Freifunk, esta plataforma foi escolhida como ambiente de testes para o projeto.

Nesse trabalho serão analisadas as métricas ETX, que estima o número de vezes que uma tentativa de transmissão deve ser repetida até que se obtenha o sucesso de transmissão, e o ETT (*Estimated Transmission Time*), que é uma evolução da métrica ETX, pois estima o tempo necessário para transmissão do pacote. Também será feita uma análise comparativa com o protocolo de número de saltos, que utiliza a distância em saltos como métrica.

Também foi implementada a métrica ETT para o `olsrd`. Entretanto, diferente do ETX que foi desenvolvido no código do programa, o ETT foi feito como *plugin*, o que o torna facilmente portável para novas versões do protocolo.

Este trabalho está dividido em três capítulos. O primeiro introduz os protocolos de roteamento ad hoc mais utilizados, com ênfase no OLSR. O segundo capítulo refere-se às métricas ETX e ETT e o terceiro à implementação. Por fim, o último refere-se a conclusão.

2. Protocolos de Roteamento

Uma rede ad hoc é uma rede sem fio descentralizada na qual os nós se comunicam diretamente uns com os outros, de forma colaborativa e sem necessitar, portanto, de uma infra-estrutura pré-existente. Para que nós que não estão dentro do alcance de rádio um do outro se comuniquem, é necessária a utilização de transmissão por múltiplos saltos. As rotas utilizadas são construídas por um protocolo de roteamento. Desta forma, nas redes ad hoc cada nó age como um roteador, de maneira a aumentar o alcance da rede, realizando a retransmissão de pacotes de outros nós colaborativamente.

Existem três tipos de protocolos de roteamento propostos para as redes ad hoc: os pró-ativos, os sob-demanda e os híbridos. Os protocolos pró-ativos se caracterizam por trocar informações sobre a rede regularmente com os demais nós de maneira a manter atualizadas suas tabelas de roteamento, de forma semelhante aos protocolos da Internet. Nos protocolos pró-ativos, as rotas estão disponíveis para uso imediato. Nos protocolos sob-demanda, ou, reativos, por outro lado, os nós requisitam informações de rota somente quando necessário, com o objetivo de economizar energia e banda passante, recursos escassos nas redes sem fio. O terceiro tipo de protocolo, o híbrido, combina características dos outros dois, onde a rede opera sob demanda ao mesmo tempo em que mantêm suas tabelas atualizadas pró-ativamente.

A seguir, serão descritos alguns dos protocolos de roteamento ad hoc mais difundidos. Serão descritos os protocolos sob-demanda DSR e AODV, o pró-ativo DSDV e o híbrido ZRP. Finalmente, será descrito o protocolo OLSR utilizado neste trabalho.

2.1. DSR

O DSR (*Dynamic Source Routing*) [8] é um protocolo ad hoc sob-demanda que utiliza roteamento pela fonte, ou seja, o nó fonte adiciona a cada pacote de dados toda a rota até o

destino. Quando todas as rotas já foram descobertas, não há *overhead* (custo adicional) com mensagens de controle e caso haja uma mudança de topologia, desde que esta não influencie o roteamento, é ignorada pelo DSR. Outra característica importante é a capacidade de armazenar múltiplas rotas para o mesmo destino, o que o diferencia de outros protocolos sob-demanda e pode ser utilizado para prover melhor qualidade de serviço, seja através da redundância ou do balanceamento de carga.

Como todos os pacotes de dados contêm o endereço de todos os nós até o destino, cada nó intermediário pode atualizar suas tabelas de roteamento com base nessa informação.

Por outro lado, a grande desvantagem do DSR também está no fato de ele carregar toda a rota nos pacotes de dados, o que acarreta um *overhead* à medida que aumenta o número de saltos, ou seja, o cabeçalho de dados se torna maior.

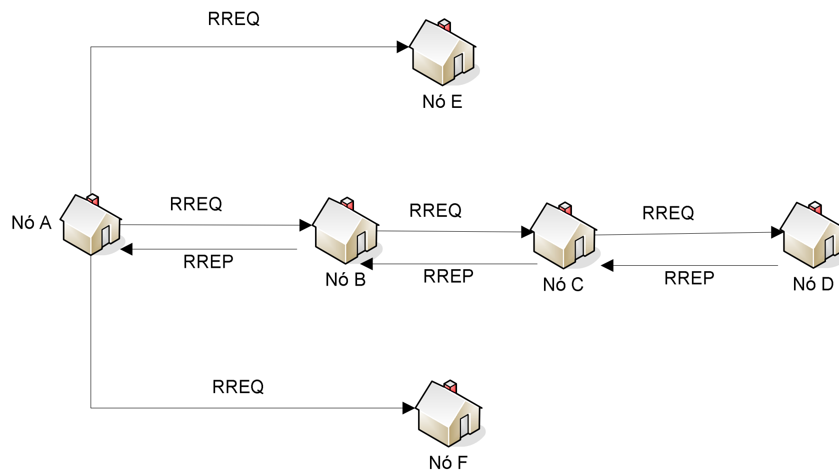


Figura 4: Descoberta de rotas no DSR.

2.1.1. Descoberta de Rotas

Quando um nó precisa de uma rota para um destino não presente na sua tabela de roteamento, ou que tenha expirado, ele efetua o procedimento de descoberta de rotas. Uma vez completado, ele encontra uma ou mais rotas para o destino, ou conclui que o destino não

está alcançável. Nesse procedimento são utilizados dois tipos de mensagens: Requisição de Rota (RREQ - *Route REQuest*) e Resposta de Rota (RREP - *Route REPly*).

Os dois tipos de mensagem são identificados pelo endereço do nó de origem, definido como endereço iniciador (*initiator address*) e pelo identificador de *broadcast* (*broadcast id*). À medida que os pacotes são encaminhados, a estação verifica a existência desse par em seu *cache*, assim como se a mensagem excedeu o tempo de expiração. Caso o tempo tenha expirado, a estação descarta o pacote, caso contrário, ela o processa. Esse procedimento evita a formação de *loops* na rede, ou seja, rotas onde uma rota passa duas vezes por um mesmo nó não alcançando o destino.

Conforme a Figura 4, o nó A inicia a inundação de uma Requisição de Rota para D e fica aguardando por uma ou mais respostas. Uma Resposta de Rota é gerada se uma estação ou mais souberem de uma rota até o destino ou se a própria estação for o destino. No caso, o nó D vai emitir uma Resposta de Rota para C e este para B e finalmente para A. Se nenhuma rota for encontrada, o nó A deve tentar novamente a Requisição de Rota mais tarde.

Cada nó que retransmite a Requisição de Rota armazena o seu identificador na mensagem retransmitida. Assim, quando a Requisição de Rota chegar ao nó D, ele terá a informação de toda a rota desse percurso, bem como cada nó intermediário terá do percurso até si, devendo então atualizarem suas tabelas de roteamento. Nesse momento D envia em *unicast* à estação A uma Resposta de Rota, que segue exatamente a rota reversa da utilizada pela mensagem de Requisição de Rota.

2.1.2. Manutenção de Rotas

O mecanismo de manutenção de rotas é executado periodicamente e entra em ação quando um nó recebe um pacote de dados para um destino não mais conhecido, ou seja, quando durante o envio de dados o enlace para o salto seguinte deixa de estar operacional.

O nó que detectou o problema constrói uma mensagem de erro de rota (RERR - *Route ERRor*) e a envia ao iniciador. Ao receber essa mensagem, o iniciador apaga essa rota e procura por outra rota no *cache* de rotas de sua tabela de roteamento. Caso não exista uma rota, ele reinicia um novo processo de descoberta de rotas.

Quando um nó intermediário recebe um pacote de erro de rota, ele também deve apagar de sua tabela a rota correspondente, inclusive para todos os nós intermediários presentes no caminho antes do ponto de falha.

No DSR, os nós ainda podem não só se utilizar das informações de rota dos pacotes sendo roteados por si, como também escutar outros pacote da rede em modo promíscuo de maneira a manter atualizada suas tabelas de roteamento. Nesse modo os nós processam também os pacotes de outros destinatários, de maneira a aproveitar alguma informação deles.

2.2. AODV

Assim como o DSR, o AODV (*Ad hoc On-demand Distance Vector*) [14] também é um protocolo sob demanda, realizando a descoberta de rotas de forma semelhante ao DSR. A principal diferença é que o AODV confia no estabelecimento dinâmico das entradas nas tabelas de roteamento dos nós intermediários, ou seja, não utiliza roteamento pela fonte para os pacotes de dados e nem *cache* de rotas. A vantagem desse processo é o menor *overhead* dos pacotes com mensagens de controle, reduzindo a sobrecarga da rede.

Entretanto, o AODV envia muito mais pacotes de controle na rede para a descoberta e manutenção de rotas do que o DSR, que envia menos, porém maiores pacotes. Isso acontece porque o DSR possui acesso a muito mais informações de rota do que o AODV, como no processo de descoberta de rotas, onde cada nó intermediário pode aprender rotas para qualquer nó ao longo da rota. Ouvir a rede promiscuamente também pode dar acesso ao DSR a muitas informações de rota. Portanto, por possuir uma quantidade de informações mais

limitada o AODV depende de mais processos de descoberta de rotas, o que acarreta uma quantidade maior de pacotes de controle na rede [9].

2.2.1. Descoberta de Rotas

Da mesma maneira que o DSR, o AODV utiliza mensagens de Requisição de Rota (RREQ - *Route REQuest*) e Resposta de Rota (RREP - *Route REPly*), sendo identificados pelo endereço iniciador e pelo identificador de *broadcast* para evitar *loops* na rede.

Quando um roteador quer obter uma rota, envia o pacote de RREQ, que inunda a rede até que algum nó que conheça uma rota até o destino, ou o próprio destino, responda com um RREP.

A mensagem de RREP contém apenas os endereços de fonte e destino, contador de saltos e número de seqüência do destino. Essas informações são usadas pelos nós intermediários que têm guardada a informação de rota do RREQ, ou seja, de quem veio essa mensagem. Com as informações do RREQ é formada a rota reversa para transmissão do RREP, além de que cada nó intermediário pode usá-la para atualizar sua tabela de roteamento. Potencialmente, pode-se economizar mensagens de controle na rede, se depois outros nós vierem a pedir uma rota para o iniciador.

Uma diferença entre o AODV e o DSR é que no primeiro, cada rota possui um tempo de expiração, uma vez que cada nó não recebe atualizações sobre a validade de suas rotas, o que ocorre no segundo caso. Assim, se a rota não for utilizada para transporte de dados após o tempo de expiração, ela é apagada. O AODV não possui a capacidade de suportar múltiplas rotas, diferente do DSR, onde toda a rota até um destino é armazenada.

2.2.2. Manutenção de Rotas

Quando um nó intermediário descobre uma falha de enlace, ou seja, não consegue encaminhar uma mensagem, ele envia ao nó de origem uma mensagem de RERR, da mesma maneira que o DSR.

Ao receber esse pacote, o nó de origem atualiza sua tabela de roteamento excluindo essa rota e inicia um novo processo de descoberta de rota. Conforme explicado no item anterior, se uma rota não for utilizada após um tempo de expiração, ela também é excluída.

2.3. DSDV

O protocolo DSDV (*Destination-Sequenced Distance Vector*) [15] é um protocolo pró-ativo, baseado no protocolo de vetor de distâncias RIP (*Route Information Protocol*) [10], que usa o algoritmo de Bellman-Ford para encontrar o menor caminho até o destino. O DSDV não é muito utilizado, sendo eficiente apenas em redes com poucos nós, tendo herdado as características principais do protocolo RIP. Desta forma, apenas será dada uma breve explicação sobre o funcionamento desse protocolo.

O problema de ocorrências de *loops* do RIP foi resolvido no DSDV através da criação de números de seqüência associados a cada mensagem o que permite identificar se uma mensagem é mais nova que outra. No DSDV, rotas com um número de seqüência maior são mais recentes e portanto preferíveis. Entretanto, isso acarreta um problema onde rotas mais novas, porém piores, podem substituir a rota antiga.

Como no RIP, cada nó envia periodicamente para os nós vizinhos sua tabela de roteamento contendo o endereço para um nó de destino, o número de saltos e o número de seqüência. Esse procedimento mantém atualizada a tabela de roteamento de todos os nós e da mesma maneira que no AODV, cada nó só conhece o próximo salto para alcançar um destino.

2.4. ZRP

O ZRP (*Zone Routing Protocol*) [11] é um protocolo híbrido. O objetivo principal é aumentar a escalabilidade da rede, dividindo-a em zonas. Dentro dessas zonas, o roteamento se dá de maneira pró-ativa e entre nós de zonas diferentes, sob demanda.

Entretanto, para a região dentro das zonas de roteamento, não é definido o funcionamento do protocolo pró-ativo a ser utilizado. Seja qual for, é definido um raio que determina a distância do centro à borda da zona, formada pelos nós periféricos.

A definição desse raio pode ser feita manual ou dinamicamente, redes com grande mobilidade são favorecidas por um raio pequeno, já redes com pouca mobilidade funcionam melhor com um raio maior, visto que nos protocolos pró-ativos enquanto não há mudança de topologia não é necessária troca de mensagens com requisição de rota, ao mesmo tempo em que todos os nós possuem rota para qualquer destino na rede.

No caso da comunicação entre nós fora da zona de roteamento, o nó faz um pedido de rota para seus nós periféricos. Caso algum nó possua rota até o destino, ele informa a rota para quem requisitou, caso contrário, esses nós periféricos irão enviar um pedido de rota para os seus respectivos nós periféricos. Esse pedido é feito sucessivamente até que algum nó periférico de uma zona de roteamento conheça uma rota até o destino.

Para evitar *loops*, esse protocolo usa um número seqüencial da mesma maneira que os protocolos anteriores. A grande vantagem do ZRP provém da mistura de características pró-ativas e reativas, porém há a introdução de uma maior complexidade de desenvolvimento e configuração.

2.5. OLSR

O OLSR (*Optimized Link-State Routing*) [5] é um protocolo pró-ativo, baseado em estados de enlace. Desta forma, periodicamente os nós inundam a rede com o estado de seus

enlaces. Assim, todos os nós podem construir um mapa completo da topologia. Como todo protocolo pró-ativo, o OLSR disponibiliza rotas imediatamente.

O que diferencia o OLSR de outros protocolos pró-ativos é que ele utiliza os chamados *MultiPoint Relays* (MPRs), que servem para diminuir o número de mensagens de controle na rede. Normalmente, quando um nó recebe pacotes de controle sobre atualizações dos estados de enlace, ele retransmite essas informações para todos os seus vizinhos, esse mecanismo é denominado inundação. Dessa maneira cada nó pode receber o mesmo pacote dos seus vizinhos diversas vezes, gerando uma grande sobrecarga de controle na rede. Esse problema ainda é agravado pelo fato de o OLSR ser um protocolo pró-ativo, ou seja, estar sempre trocando informações. O objetivo dos MPRs é minimizar esse problema através da seleção de nós que irão fazer a inundação.

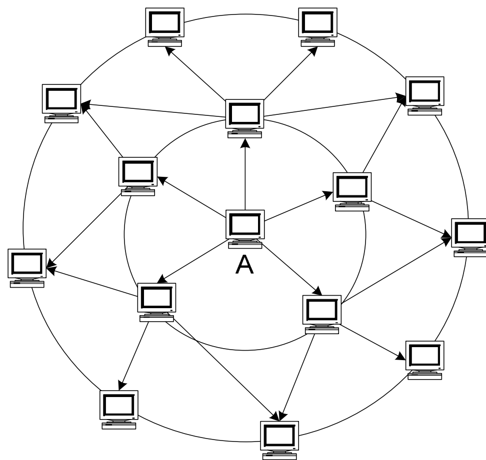


Figura 5: Inundação normal.

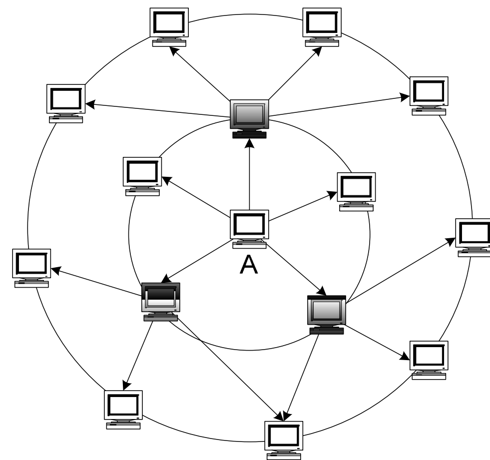


Figura 6: Inundação com MPR.

Cada nó da rede escolhe os seus MPRs, ou seja, nós designados para retransmitir os pacotes de controle. A escolha de um nó como MPR é baseada na premissa de que o nó consiga alcançar todos os seus vizinhos de dois saltos através do menor número de MPRs possível. Ou seja, através dos MPRs o nó de origem deve alcançar qualquer nó a dois enlaces de distância, de maneira que todos esses recebam as mensagens de controle da origem.

As figuras ilustram o funcionamento desse tipo de rede e sua alcançabilidade. No exemplo da Figura 5, é feita a inundação normal sem o uso de MPR, já na Figura 6 somente os nós designados, que são os MPRs do nó "A" (identificados em negrito) irão retransmitir os pacotes de inundação.

Assim, como cada nó possui seus MPRs designados, a comunicação entre quaisquer nós da rede é feita pelos MPRs dos nós intermediários. Essa otimização faz com que esse protocolo produza menor sobrecarga de controle, principalmente para redes grandes e de alta densidade, sendo por outro lado melhor aplicável àquelas redes com pouca mobilidade, onde a topologia permaneça constante uma vez que não é gerado nenhum controle adicional de tráfego.

A RFC 3626 [5], que especifica o protocolo OLSR, detalha todos os componentes para o funcionamento do protocolo. A seguir serão descritos somente os componentes essenciais para o entendimento deste projeto, a saber: o formato do pacote, a mensagem de HELLO, o cálculo do MPR, a descoberta de topologia e o cálculo da tabela de roteamento. São ainda definidos componentes para suporte a múltiplas interfaces (MID) e divulgação de *gateways* para a Internet (mensagens HNA). Entretanto, por não apresentarem relevância para a proposta deste trabalho, eles não serão detalhados.

2.5.1. Formato do Pacote

O OLSR tem todos os seus pacotes de controle enviados sobre UDP (*User Datagram Protocol*), utilizando a porta de número 698, concedida pela IANA [12]. O formato básico dos pacotes do OLSR omitindo cabeçalhos IP e UDP é descrito na Figura 7.

O campo `tamanho do pacote` representa o tamanho do pacote todo, o `número de seqüência do pacote` é um número que deve ser incrementado de um toda vez que um pacote é transmitido, sendo utilizado apenas para a detecção de perda de pacotes.

O campo `tipo da mensagem` representa dentro do protocolo o tipo de mensagem que é transmitido, como por exemplo, as mensagens de HELLO e mensagens TC, que correspondem aos tipos 1 e 2, respectivamente. Conforme será mostrado mais adiante, as mensagens do *plugin* desenvolvido nesse projeto são identificadas por um número local ou personalizado, que é a faixa de valores entre 128 e 255.

O campo `Vtime` representa o tempo de validade dos dados contidos na mensagem, caso o nó não venha a receber nenhuma nova mensagem com atualizações. Os campos seguintes são o tamanho da mensagem desde o campo `tipo da mensagem` até o fim do pacote ou o início da mensagem seguinte, o endereço IP de quem gerou a mensagem e o tempo de vida.

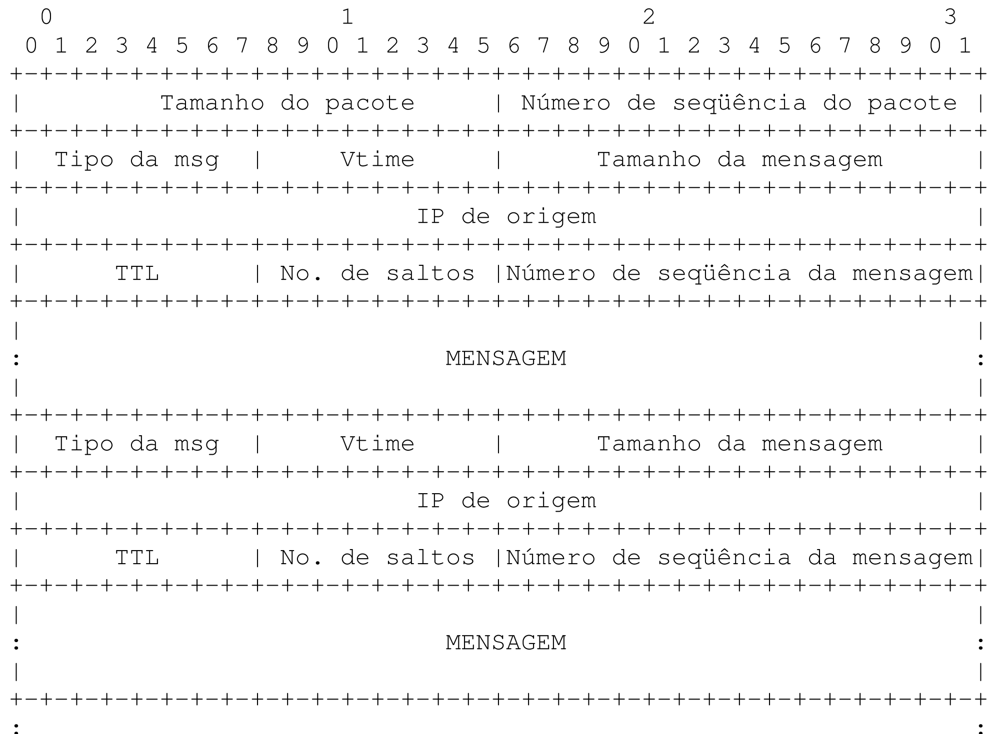


Figura 7: Formato do pacote do OLSR.

O campo `número de saltos` deve ser incrementado por todo nó antes de ser reencaminhado e representa a distância em saltos percorrida pelo pacote. O campo de `número`

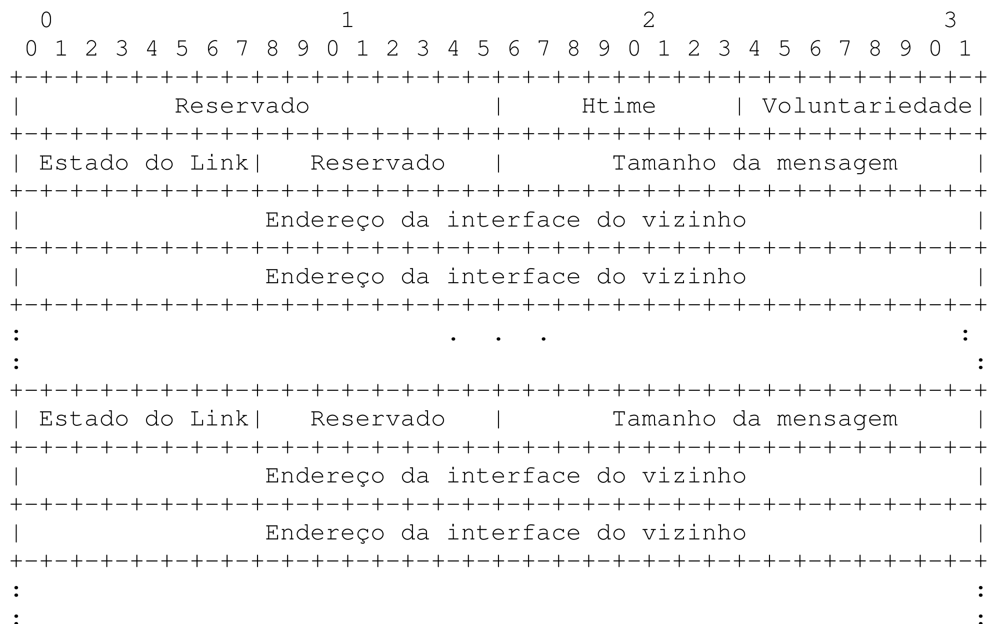
de seqüência da mensagem tem por objetivo evitar que uma mensagem seja processada duas vezes pelo mesmo nó.

O campo MENSAGEM contém as mensagens que são transmitidas pelo pacote do OLSR, onde cada pacote pode ter concatenado várias mensagens de maneira a diminuir a sobrecarga com pacotes de controle na rede.

2.5.2. Mensagem de HELLO

A mensagem de HELLO tem por objetivo tanto a descoberta de vizinhos e a sinalização da seleção de MPRs, bem como a verificação da conectividade do enlace aos vizinhos. Essa mensagem não deve ser reencaminhada pelos nós, por isso seu TTL é colocado em 1.

O formato da mensagem de HELLO, que corresponde ao campo MENSAGEM do pacote na Figura 7 é mostrado na Figura 8.



```
  0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Reservado            |     Htime     | Voluntariedade|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Estado do Link|   Reservado   |       Tamanho da mensagem     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Endereço da interface do vizinho               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Endereço da interface do vizinho               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                             . . .                           :
:                                                               :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Estado do Link|   Reservado   |       Tamanho da mensagem     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Endereço da interface do vizinho               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Endereço da interface do vizinho               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                                                               :
:                                                               :
```

Figura 8: Formato da mensagem de HELLO.

O campo `Htime` especifica o intervalo de emissão desse tipo de mensagem. O campo `voluntariedade` especifica se o nó pode (WILL_DEFAULT), não pode (WILL_NEVER) ou sempre (WILL_ALWAYS) será selecionado como MPR.

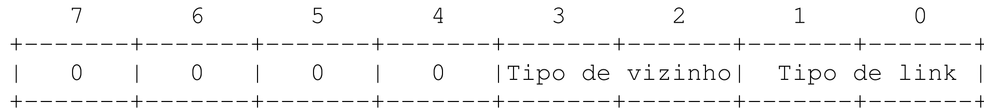


Figura 9: Estado do enlace.

O campo `estado do link` é divido conforme mostrado na Figura 9, onde o `tipo de link` indica para cada enlace do emissor do HELLO com um vizinho, se ele é simétrico (SYM_LINK), assimétrico (ASYM_LINK) ou se caiu (LOST_LINK). Um enlace simétrico significa que a comunicação foi estabelecida e confirmada em ambas as direções. Um enlace assimétrico indica que o emissor do HELLO é capaz de ouvir mensagens do vizinho, mas não sabe se suas mensagens são ouvidas por ele.

O `tipo de vizinho` indica, para cada enlace com um vizinho do nó que enviou o HELLO, que pelo menos uma interface desse vizinho tem conexão simétrica com o emissor (SYM_NEIGH), ou ainda que o enlace é simétrico e selecionado como seu MPR (MPR_NEIGH), ou é assimétrico ou indisponível (NOT_NEIGH).

Ao receber uma mensagem de HELLO, o nó deve atualizar sua tabela de enlaces, ou seja, uma tabela que contém os vizinhos distantes um salto, incluindo as informações sobre o tipo de enlace e validade daquela informação (campo `Vtime`).

Da mesma maneira, o nó também deve atualizar a tabela de vizinhos distantes um salto incluindo a informação relativa à sua voluntariedade (ser MPR ou não). Note que essa tabela é parecida com a anterior, já que ambas funcionam em par. Entretanto, a primeira guarda informações relativas ao estado do enlace, enquanto que a segunda sobre o vizinho.

No caso desse vizinho ter selecionado este nó como MPR, o que é indicado pelo campo `tipo de vizinho` igual a MPR_NEIGH, ele deve ser adicionado à lista de selecionador MPR. Essa lista identifica que nós selecionaram este nó como MPR.

O nó deve processar também as informações relativas aos nós distantes dois saltos dele, ou seja, aqueles com os quais seus vizinhos possuem alcance direto, mas com os quais ele não possui. Assim, para cada endereço listado no campo `endereço da interface do vizinho`, o nó deve checar se este não é o próprio e adicioná-lo à tabela de vizinhos distantes de dois saltos, incluindo o endereço do vizinho emissor do pacote e o tempo de validade da informação.

Em todas as tabelas, de enlaces, de vizinhos distantes de um salto e de vizinhos distantes de dois saltos, a informação é excluída da tabela caso o seu tempo de validade expire ou o campo `estado do link` da mensagem de HELLO indique que o enlace deixou de existir ou, ainda no último caso, que se tornou assimétrico.

A construção da mensagem de HELLO é feita de maneira análoga à recepção, onde cada nó deve enviar as informações referentes à sua tabela de enlaces, de vizinhos e de seus MPRs.

2.5.3. Cálculo do MPR

Conforme mencionado anteriormente, os MPRs de um nó têm que ser calculados de maneira que todos os vizinhos distantes dois saltos sejam atingidos por pelo menos um MPR. A escolha dos MPRs por um nó se dá de maneira independente dos outros nós da rede. São utilizados apenas enlaces simétricos, que são anunciados aos demais nós através do campo `tipo de vizinho` presente na mensagem de HELLO.

É interessante notar que caso todos os nós sejam selecionados como MPR, a inundação ocorrerá como em um protocolo de estados de enlace normal. Um exemplo para esse caso ocorre na inicialização do protocolo.

O algoritmo para a escolha dos MPRs é executado para cada interface de rede física, sendo os MPRs de um nó a união dos MPRs de todas as interfaces. Toda vez que for detectada a entrada ou saída de um nó a até dois saltos de distância, o algoritmo de escolha dos MPRs deve ser executado novamente.

Para o algoritmo de escolha dos MPRs, considera-se N como sendo a lista de vizinhos de uma interface I de um nó e N^2 todos os nó alcançáveis de I por algum N, exceto: aqueles com campo `voluntariedade` da mensagem de HELLO identificados por WILL_NEVER, ou seja, que nunca serão selecionados como MPR; o nó efetuando o cálculo (definido por I) e todos os nós em que existe um enlace simétrico com apenas um salto de distância de I. Ou seja, N^2 representa o conjunto de vizinhos de dois saltos do nó pela interface I. Por fim, considera-se D(y) como sendo o número de vizinhos para cada nó y pertencente a N, exceto pelo nó efetuando o cálculo (I) e pelos membros de N.

O algoritmo é efetuado da seguinte maneira:

1. Comece com a lista de MPR apenas com os nós cujo campo voluntariedade da mensagem de HELLO seja WILL_ALWAYS, ou seja, que desejam sempre ser selecionados como MPR;
2. Calcule D(y) para cada nó y em N;
3. Adicione à lista de MPRs aqueles nós em N que são os únicos através dos quais é possível alcançar um nó em N^2. Remova esse nó de N^2 que agora é coberto por um novo MPR;
4. Enquanto existirem nós em N^2 que não forem cobertos por pelo menos um MPR:
    a. Para cada nó y em N, calcule o número de nós em N^2 que não forem cobertos por pelo menos um MPR e que forem alcançáveis por ele;

b. Adicione à lista de MPRs o nó que alcançar mais nós em N^2 e em caso de empate, aquele que alcançar mais nós de segunda ordem, ou seja, maior D(y). A seguir, remova os nós em N^2 que agora são cobertos pelo novo MPR;

5. O conjunto de MPRs do nó será a união da lista de MPRs de cada interface.

Assim, no caso de redes com mudanças de topologia constantes, como por exemplo, causadas por maior mobilidade, é recomendado que se tenha um conjunto maior de MPRs redundantes para evitar o recálculo desse algoritmo.

2.5.4. Descoberta de Topologia

Os mapas de topologia são construídos através da propagação das informações relativas aos vizinhos obtidas dos pacotes de HELLO e pelas mensagens de controle de topologia (*Topology Control Messages* – TC *messages*).

Essas mensagens, cujo formato está mostrado na Figura 10, são disseminadas pelos nós MPR, contendo informações suficientes para que cada nó construa suas tabelas de roteamento.

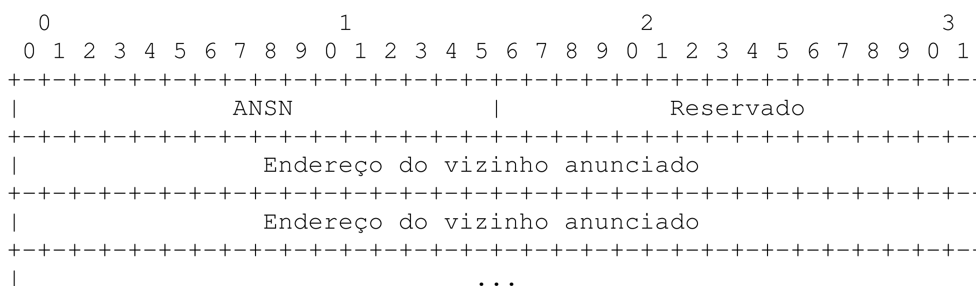


Figura 10: Mensagem de controle de topologia.

O campo `ANSN` (*Advertised Neighbor Sequence Number*) representa o número de seqüência do anúncio. Toda vez que alguma mudança de topologia for detectada, esse número deve ser incrementado. Esse mecanismo permite que os demais nós saibam qual é a mensagem mais recente sobre a topologia da rede.

Cada campo `Endereço do Vizinho Anunciado` contém o endereço de um nó vizinho ao emissor do pacote. Esse nó deve ser distante um salto e devem ser enviados na mensagem pelo menos os endereços dos nós selecionados como MPRs do nó originador do pacote.

A mensagem de controle de topologia deve ser enviada com TTL de valor máximo e deve ser propagada de maneira a atingir todos os nós da rede. Através da lista de vizinhos contidos no pacote, cada nó obtém todas as informações necessárias para construir sua tabela de roteamento. Mesmo quando a tabela de vizinhos estiver vazia, o nó deve ainda assim enviar a mensagem TC de forma a que os demais nós possam atualizar suas tabelas de roteamento.

Ao receber uma mensagem TC, caso o enlace não exista na tabela de topologia, o nó deve adicionar para cada vizinho anunciado uma entrada contendo o endereço do emissor do pacote, o número de seqüência (ANSN) e o tempo de validade da informação.

A mensagem TC pode conter como redundância outros nós que não os MPRs, o que ajudaria a criar outras rotas para o tráfego de dados. Existem três parâmetros de configuração definido: o primeiro, no qual só os endereços dos MPRs são incluídos nos vizinhos anunciados, o segundo, onde além dos MPRs, os nós selecionados para serem MPR de outros nós são enviados e, finalmente, o terceiro modo de operação onde todos os vizinhos são incluídos.

2.5.5. Cálculo da Tabela de Roteamento

A tabela de roteamento é construída com base nas informações contidas na tabela de enlaces e na tabela de topologia, sendo reconstruída quando uma dessas tabelas sofre uma mudança, ou ainda quando uma das tabelas de vizinhos distantes de um salto e de vizinhos distantes de dois saltos sofrem mudança.

Tabela 1: Formato da tabela de roteamento.

Destino	Próximo Salto	Distância em Saltos	Interface de Saída
R_dest_addr	R_next_addr	R_dist	R_iface_addr
R_dest_addr	R_next_addr	R_dist	R_iface_addr
...	...	...	...

Cada campo da tabela de roteamento, conforme a Tabela 1, tem o seguinte significado:

- R_dest_addr: Endereço do destino da rota;
- R_next_addr: Endereço do próximo salto;
- R_dist: Distância em saltos até o destino;
- R_iface_addr: Interface de saída da rota.

Para a construção da tabela de roteamento, os nós distantes de um e de dois saltos são adicionados com base nas informações das mensagens de HELLO. Já para os outros nós, essas informações são baseadas na tabela de topologia construída a partir das mensagens de Controle de Topologia com base nas informações de distância, endereço do nó de origem da mensagem e do vizinho anunciado.

Assim, o seguinte procedimento é utilizado para a construção da tabela de roteamento:

1. Todas as entradas da tabela de roteamento são removidas;
2. São adicionadas entradas para os vizinhos simétricos distantes de um salto, atribuindo R_dist igual a um;
3. Para cada nó N^2, é adicionada uma rota com R_dest_addr apontando para o endereço desse nó, com R_dist igual a dois e R_next_addr sendo o endereço do MPR que leva a ele.
4. Uma nova entrada é adicionada para cada nó a $h + 1$ saltos de distância. O seguinte procedimento deve ser executado para cada valor de h, sendo terminado se nenhuma nova entrada for adicionada na iteração.
    a. Para cada entrada da tabela de topologia, verificar se o seu endereço de origem corresponde a alguma entrada de R_dest_addr com R_dist igual a h. Caso corresponda, adicionar uma nova entrada, com R_dest_addr

igual ao endereço do vizinho anunciado, R_next_addr com o endereço R_next_addr da entrada correspondente encontrada na tabela. E R_dist igual a $h + 1$.

Muitos nós podem ser utilizados como endereço do próximo salto (R_next_addr) para alcançar um destino (R_dest_addr), entretanto aqueles selecionados para serem MPR de outro nó são preferíveis como próximo salto.

Observe que por causa da forma como a sua rota é calculada, não é possível ter múltiplas rotas para um mesmo destino, uma vez que a cada vez que for calculada a tabela de roteamento, apenas uma rota para cada destino será obtida.

3. Métricas de Roteamento

3.1. ETX

A métrica ETX (*Expected Transmission Count*) visa minimizar o número esperado de tentativas de transmissão para uma transmissão com sucesso, incorporando os efeitos de taxa de perda, assimetria nas taxas de perda nas duas direções de um enlace e interferência ao longo dos sucessivos enlaces de um caminho. Nos protocolos mencionados anteriormente, e na maioria dos protocolos ad hoc encontrados na literatura, o melhor caminho é aquele que minimiza a métrica número de saltos, a despeito da vazão em todos os caminhos possíveis.

Utilizando simplesmente a métrica número de saltos, os protocolos assumem que cada enlace ou está funcionando ou está inoperante e neste caso simplesmente não há nenhuma comunicação. Embora essa suposição seja muito próxima da realidade para redes cabeadas, não é o caso das redes sem fio. Enlaces com uma elevada taxa de perda de pacotes dados, entregando menos de 50% dos pacotes, mas ainda sim transportando mensagens de controle, seriam considerados da mesma maneira que enlaces com boa vazão.

A proposta da métrica ETX é encontrar caminhos que produzam o menor número necessário de retransmissões. Para isso a métrica prevê o número de retransmissões esperadas fazendo medições da taxa de perda de pacotes para cada enlace da rede. O objetivo é, assim, encontrar rotas com a melhor vazão.

Para tanto, se o ETX de um enlace é o número de transmissões necessárias para enviar um pacote sobre ele, o ETX de uma rota será a soma dos ETX de cada enlace.

Em um teste descrito em [1], usando ETX com os protocolos DSR e DSDV, foram obtidas vazões melhores por uma razão de dois com o uso do ETX. Os resultados apresentados são melhores à medida que o número de saltos aumenta, mostrando a eficácia da métrica conforme a rede cresce.

No caso do protocolo OLSR a solução adotada foi diferente da vista em [1], uma vez que para medir a taxa de erro foi usado o próprio pacote de HELLO que já é transmitido periodicamente na rede, ao invés de se criar um outro tipo de mensagem, evitando assim o aumento da sobrecarga de controle da rede. Uma desvantagem de seu emprego, é que sempre será enviado um pacote de tamanho pequeno, uma vez que o pacote de HELLO possui essa característica.

Como cada nó sabe que os pacotes de HELLO são transmitidos periodicamente (a cada dois segundos na configuração padrão), ele pode medir a taxa de erro relacionando a quantidade de pacotes recebidos com a quantidade de pacotes esperados em uma determinada janela de tempo.

Por exemplo, se 3 em cada 10 pacotes de HELLO não forem recebidos pelo vizinho, tem-se uma taxa perda de 3/10 = 30%. Assim, como 7 pacotes chegaram tem-se uma qualidade de enlace (*Link Quality* – LQ) de 70%.

Entretanto, como os enlaces são bidirecionais, cada nó também obtém informações sobre a qualidade do enlace relativa aos pacotes que ele está enviando, ou seja, a visão que os vizinhos têm de seu enlace (*Neighbor Link Quality* – NLQ).

Como ambos esses valores são enviados em percentagem, eles representam a probabilidade de um pacote atravessar um enlace com sucesso em cada direção. Assim, a probabilidade de sucesso de uma transmissão será o produto dessas: LQ x NLQ.

Pode-se concluir então que o número de tentativas esperado para que um pacote possa ser transmitido com sucesso é:

$$\mathrm{ETX} = \frac{1}{\mathrm{LQ} \times \mathrm{NLQ}}$$

É importante notar que esse valor é válido nos dois sentidos do trajeto, uma vez que haverá retransmissão tanto se o pacote de dados quanto se o respectivo ACK forem perdidos.

Finalmente, conforme mencionado anteriormente, o ETX de uma rota será a soma da métrica de cada enlace dos nós intermediários.

Para incluir a informação de qualidade do enlace nas mensagens de HELLO do OLSR, elas foram modificadas como proposto na RFC 3626 [5] e mostrado anteriormente. Conforme a Figura 11, a nova mensagem chamada de LQ HELLO contém agora dados sobre a qualidade do enlace.

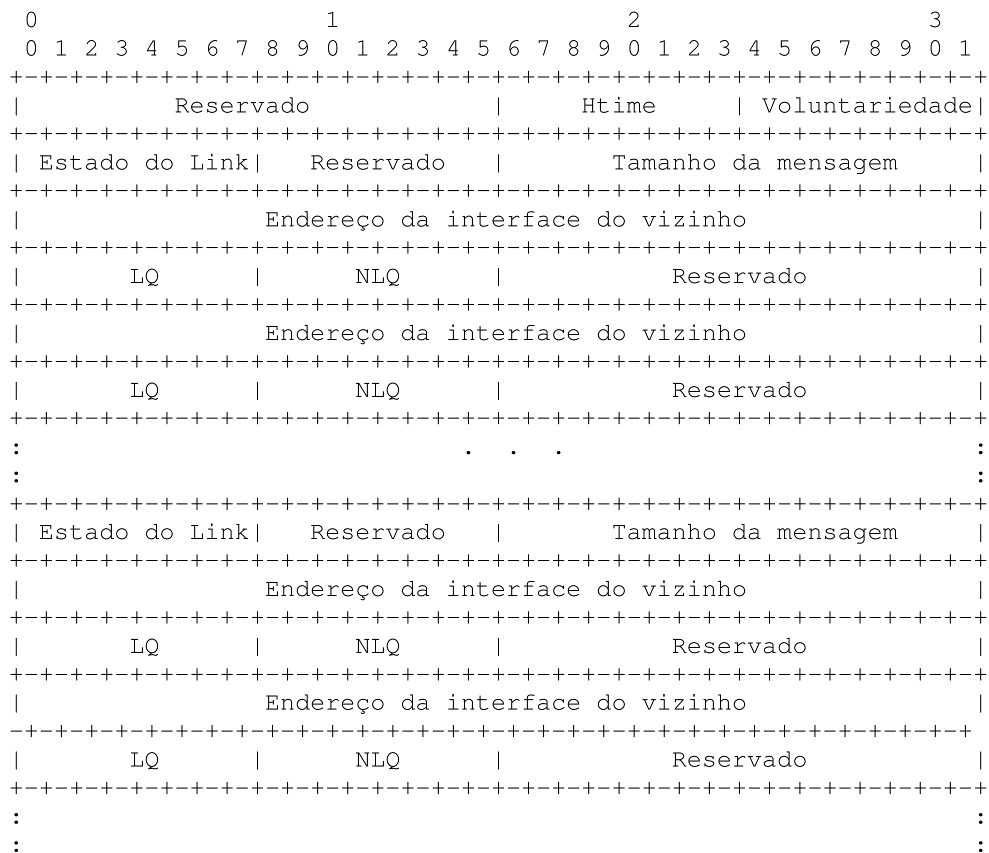


Figura 11: Mensagem LQ HELLO.

Observe que após o endereço do vizinho foi incluída a informação relativa à qualidade do enlace nos dois sentidos.

Para transmitir essa informação para todos os vizinhos para que eles a considerem no cálculo da rota, também foi necessário alterar a mensagem de controle de topologia, que foi denominada LQ TC, e cujo formato é apresentado na Figura 12.

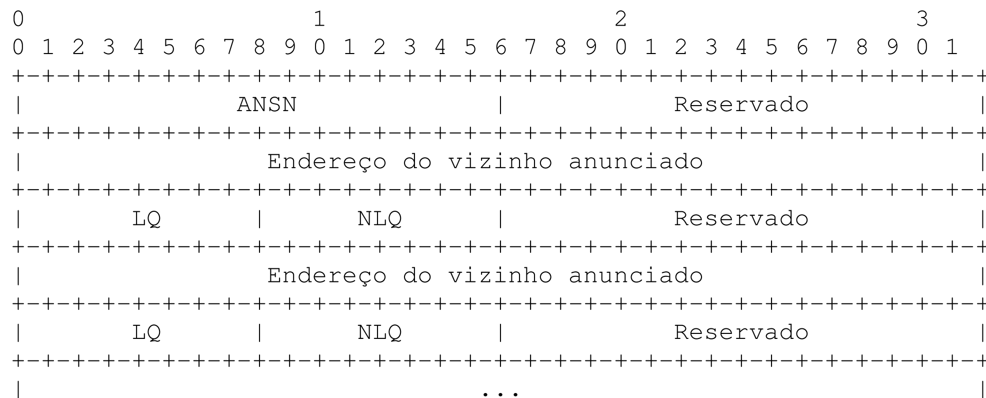


Figura 12: Mensagens LQ TC.

Da mesma maneira que na LQ HELLO, informações relativas à qualidade do enlace do nó que está enviando a mensagem com o respectivo vizinho são incluídas logo após seu endereço.

Para a construção da tabela de roteamento utilizando a qualidade da métrica é empregado o algoritmo de Dijkstra [16], onde é procurado em toda a topologia o caminho de menor custo entre dois nós.

3.2. ETT

Uma grande desvantagem da métrica ETX é levar em conta apenas a taxa de perdas em um dado enlace, para um tamanho pequeno de pacote. Uma métrica proposta para descrever de forma mais precisa a qualidade do enlace é a ETT (*Expected Transmission Time*) [2], que leva em consideração também o tempo de transmissão do pacote em cada enlace.

Em [2], foi desenvolvida uma solução baseada no protocolo LQSR, onde essa métrica foi testada e aplicada. Neste trabalho, foi também proposta uma combinação dessa métrica onde são utilizados múltiplos rádios, entretanto como no nosso trabalho só foi testado e implementado o ETT, só nos ateremos a esse ponto.

Uma forma de calcular a métrica ETT é utilizar o valor do ETX, obtido como explicado anteriormente, e multiplicá-lo pelo tamanho do pacote dividido pela banda passante

do enlace, de maneira a obter o tempo esperado de transmissão do pacote. Assim, se S for o tamanho do pacote e B a banda do enlace teremos:

$$ETT = ETX \times \frac{S}{B}$$

Para obter uma estimativa da banda passante do enlace, foi usado o método de pares de pacote, onde a cada minuto um nó envia um par de pacotes para os seus vizinhos. Nesse método, implementado em [2], esses pacotes são enviados em *unicast*. O primeiro pacote é pequeno, contém 137 bytes e o segundo é grande, de 1137 bytes. Quando um nó recebe esse par de pacotes, ele calcula a diferença de tempo e comunica esse valor de volta ao emissor. A banda é calculada após coletadas 10 amostras consecutivas e dividindo o tamanho do segundo pacote pela menor dessas amostras.

Essa medida pode não ter muita acurácia, mas é suficiente para distinguir a qualidade de enlaces bons e ruins. Conforme resultados obtidos em [2], para taxas mais baixas, a precisão é maior, mas para mais altas o resultado subestima a taxa real de transmissão. Esse problema é causado devido ao fato de que os *overheads* envolvidos na transmissão do pacote, como tempo para enviar um ACK, se tornam mais significativos à taxas mais altas.

De acordo com os resultados obtidos em [2], em ambientes com taxa de transmissão variável o ETT apresenta um desempenho 16% superior à métrica ETX e 38,6% superior à métrica número de saltos. Vale ressaltar de que essa implementação não foi idêntica a desse trabalho, cujos pacotes são enviados em *broadcast* e utilizando outra métrica.

Não existia, até o presente momento, para o OLSR nenhuma implementação utilizando a métrica ETT. Por isso foi desenvolvida, nesse trabalho, uma implementação de tal métrica. Essa solução é um *plugin* para o programa `olsrd` que conforme já dito, foi o *software* adotado.

Nesse *plugin*, conforme ainda será detalhado no capítulo seguinte, foi reproduzida a implementação descrita em [2], entretanto com os pacotes enviados em *broadcast*, para a métrica ETT com apenas uma interface.

4. Implementação da Métrica ETT no OLSR

4.1. A Plataforma

A plataforma de *hardware* utilizada nesse trabalho é composta de microcomputadores, e por roteadores sem fio Cisco Linksys modelo WRT54g (Figura 13). Esses roteadores são conhecidos por serem os primeiros a terem o código fonte de seu *firmware* divulgado para atenderem as obrigações da GNU GPL. Isso motivou programadores a entenderem a arquitetura do equipamento e permitiu que fossem desenvolvidos diversos sistemas operacionais para ele. É nesse contexto que se deu a escolha desse *hardware* para a implementação da métrica ETT, pois além de ser um dispositivo dedicado possui a característica de ser mais facilmente posicionado. Outra vantagem é o baixo custo e independência em relação ao uso de microcomputadores, já que os testes não ficam condicionados à sua disponibilidade.

Figura 13: Linksys WRT54g.

A Cisco lançou várias versões desse dispositivo com diferentes configurações de *hardware*, algumas mais recentes inclusive com o objetivo de limitar o uso de outros sistemas operacionais em seus aparelhos. Os dispositivos usados no ambiente de testes foram nove

roteadores de versão 2.2 e um roteador versão 5.0 cujas configurações se encontram na Tabela 2.

Tabela 2: Configurações dos roteadores do ambiente de testes.

Versão	CPU	Memória RAM	Memória flash
2.2	216 MHz	16 MB	4 MB
5.0	200 MHz	8 MB	2 MB

Como pode ser observado, a Versão 5.0 apresenta claras limitações de memória em relação a Versão 2.2, além de apresentar diferenças de arquitetura. Por isso a distribuição do Linux a ser usada na versão 2.2 é o OpenWRT, que não é suportado pela versão 5.0.

O Linksys usa plataforma MIPS (*Millions of Instructions Per Second*), ou seja, com um processador RISC e pode suportar dois sistemas de arquivo em sua memória *flash*, o SquashFS e o JFFS2. O SquashFS apresenta a característica de ser somente para leitura, entretanto apresenta uma melhor compressão dos arquivos e acesso mais rápido. O sistema JFFS2 admite escrita, porém lenta e volumosa.

A configuração usada nos Linksys versão 2.2 será o OpenWRT RC5 (*Release Candidate 5*) com sistema de arquivos SquashFS, entretanto com uma partição JFFS2 de escrita, de maneira que se possa editar arquivos de configuração sem a necessidade de reescrever o *firmware*.

Já no caso do Linksys versão 5.0 foi usada a distribuição micro do DD-WRT v23 SP2 que com o mínimo de recursos permite que um sistema menor que 2 MB e somente leitura (SquashFS) seja instalado no roteador.

4.1.1. Compilando para a Plataforma

Por ser uma plataforma diferente da usada nos microcomputadores, o modo de compilação também não é o mesmo. É preciso utilizar um outro compilador e executar algumas mudanças no código para que qualquer programa possa ser compilado para a arquitetura MIPS.

Para isso foi utilizado o *OpenWrt Software Development Kit* (SDK) que permite compilar aplicações facilmente para a plataforma, bastando a criação de apenas alguns arquivos de configuração que seguem no Apêndice 7.2.

Esses arquivos configuram como o pacote será gerado. O principal deles é o Makefile, que define um local para baixar o arquivo e o compila gerando o *plugin.* A seguir, o Makefile o usa para construir o pacote.

4.2. Código Fonte

Foi escolhido desenvolver um *plugin* para implementar a métrica ETT no `olsrd`. A grande vantagem dessa escolha é de não haver necessidade de modificar o código, privilegiando desta forma a portabilidade.

Conforme já foi dito, o `olsrd` já vem com a métrica ETX implementada em seu código, então tudo que o *plugin* faz é medir a banda, calcular o valor modificador de métrica e modificá-la.

Para efetuar essa modificação de métrica é aproveitado o parâmetro do arquivo de configuração *LinkQualityMult* que altera estaticamente o LQ de um enlace para um determinado nó, ou seja, em seu funcionamento normal pode-se atribuir pesos a determinados nós no arquivo de configuração de maneira a privilegiá-los na escolha de rota. O *plugin* modifica sua estrutura dentro do programa de maneira que a alterar seus parâmetros dinamicamente, ou seja, conforme é feita a medição do tempo de transmissão esse valor que atribui pesos aos nós é alterado.

Para realizar a medição de banda conforme já foi descrito, foram criados dois novos tipos de mensagem, um chamado de *ett_small_olsrmsg* e outro chamado de *ett_big_olsrmsg*. O trecho a seguir, retirado do código fonte, no Apêndice 7.1, exibe o formato das estruturas desses pacotes.

```
/*
 * Packet message definition
 */

struct ettmsg
{
  olsr_u32_t    destination[83];
  double        ett_time[83];
};

/*
 * OLSR messages
 */

struct ett_big_olsrmsg
{
  olsr_u8_t      olsr_msgtype;
  olsr_u8_t      olsr_vtime;
  olsr_u16_t     olsr_msgsize;
  olsr_u32_t     originator;
  olsr_u8_t      ttl;
  olsr_u8_t      hopcnt;
  olsr_u16_t     seqno;

  struct ettmsg msg;
  char          getspace[BIG_PKT_SIZE-1012];
};

struct ett_small_olsrmsg
{
  olsr_u8_t      olsr_msgtype;
  olsr_u8_t      olsr_vtime;
  olsr_u16_t     olsr_msgsize;
  olsr_u32_t     originator;
  olsr_u8_t      ttl;
  olsr_u8_t      hopcnt;
  olsr_u16_t     seqno;

  char                  getspace[SMALL_PKT_SIZE-12];

};
```

O primeiro tipo de pacote leva em sua mensagem 83 espaços para endereço seguido de 83 espaços para a diferença de tempo. O endereço refere-se a quem enviou o par de diferença de tempo, de maneira que o nó que está recebendo a mensagem saiba qual foi a diferença de tempo enviada em seu último pacote, e pelo cabeçalho da mensagem de quem veio. Conforme pode ser visto nos códigos fonte, cada espaço da variável de endereço (*destination*) possui 4 bytes e cada espaço de tempo (*ett_time*), 8 bytes, o que totaliza 996 bytes, que junto com o cabeçalho e o *payload* somam 1137 bytes.

O segundo tipo de pacote contém o cabeçalho normal dos pacotes do protocolo e um *payload* (`getspace`) no campo de mensagem para que o tamanho da mensagem seja de 137 bytes definido pela variável `SMALL_PKT_SIZE`.

O programa possui duas estruturas principais necessárias ao seu funcionamento. A primeira, chamada de *ett_entry*, contém as dez últimas diferenças de tempos de cada nó do qual foi recebido um pacote, sendo essas atualizadas de maneira deslizante, ou seja, a informação mais antiga é atualizada. Conforme pode ser visto a seguir, estão incluídos nela, o endereço do vizinho, o temporizador de validade da informação, uma matriz com dez diferenças de tempo e a referência que aponta qual o valor mais recente nessa matriz. A segunda estrutura, chamada de *ett_compute_time*, tem como finalidade guardar a hora de recebimento de cada pacote e o endereço de quem o enviou, de maneira a calcular e enviar de volta a diferença de tempo de recebimento dos pacotes. A seguir é mostrado o trecho do código fonte onde essas estruturas são definidas:

```
/* Database entry */

struct ett_entry /* Fill this structure upon receipt of probe with data I
sent */
{
        union olsr_ip_addr     neighbour;  /* IP address of the neighbour
that sent the probe */
        struct timeval         timer;      /* Validity time */
        double                             ett_time[SAMPLES_MAX];   /* List of
time difference to the neighbour */
        short unsigned         last_used;      /* Last sample used */
        struct ett_entry       *next,*prev;
};

struct ett_compute_time /* Fill this struture upon receipt of probe with
time each neighbour sent me wants to know */
{
        union olsr_ip_addr     originator;  /* IP address of the neighbour
*/
        struct timeval         big_probe_time, small_probe_time;   /* Time
between probes is the difference (last par) */
        struct ett_compute_time        *next,*prev;
};
```

Ao receber um par de pacotes o nó deve executar dois procedimentos. No primeiro, após receber o primeiro pacote, ele preenche a estrutura *ett_compute_time* junto com o

endereço do emissor e a hora corrente, e logo após preenche com a hora do segundo pacote. O segundo procedimento visa separar as informações referentes à diferença de tempo enviada pelo seu último par de pacotes, caso haja. Assim, o nó procura no pacote *ett_big_olsrmsg* pelo seu endereço e pelo tempo correspondente adicionando essa diferença de tempo na estrutura *ett_entry*.

É importante frisar que a tomada de tempo é feita no início das funções, de maneira que não seja levado em conta o tempo que se gasta executando os procedimentos.

O intervalo de emissão (*emission_interval*) em que o *plugin* envia o par de pacotes é feito a cada 4,5 segundos por padrão, que entretanto não foi o tempo utilizado nos testes. Os pacotes possuem TTL em 1, de maneira que o pacote não é encaminhado pelos próximos nós, entretanto o *plugin* não gera mensagem de erro ICMP. Eles são enviados para o endereço de *broadcast*, bem como todo pacote de controle gerado pelo `olsrd`, conforme visto na seção 2.5.1. Durante a construção desse pacote o nó percorre a estrutura *ett_compute_time* e calcula a diferença de tempo dos pacotes, incluindo-os na mensagem e enviando-a. Nesse momento a entrada também é apagada da estrutura, uma vez que já foi utilizada, porém naquelas entradas que ainda não tiver o segundo pacote nada é feito.

Os dados na estrutura *ett_entry* têm um tempo de validade (*ett_expiration_time*) de 20 segundos por padrão, ou seja, se em um espaço de 20 segundos nenhum pacote contendo diferença de tempo em relação a um determinado nó for recebido, a sua entrada na estrutura é apagada. Esse procedimento é adotado porque caso um nó saia da rede, sua informação não teria outra maneira de ser apagada.

Para registrar junto ao programa `olsrd` os procedimentos descritos no *plugin*, são especificadas na função de inicialização, *olsrd_plugin_init*, as funções que o *plugin* irá chamar. Isto é exemplificado no trecho de código abaixo:

```
int
olsrd_plugin_init()
{
```

```
...

  /* Register functions with olsrd */
  olsr_parser_add_function(&olsr_parser_big_probe, BIG_PARSER_TYPE, 1);
  olsr_parser_add_function(&olsr_parser_small_probe, SMALL_PARSER_TYPE, 1);

  olsr_register_timeout_function(&olsr_timeout);

  olsr_register_scheduler_event(&olsr_event, NULL, EMISSION_INTERVAL, 0,
NULL);

  return 1;
}
```

A função *olsr_parser_add_function* adiciona a função de particionamento de pacotes, ou seja, quando um pacote do tipo definido, no caso BIG_PARSER_TYPE ou SMALL_PARSER_TYPE, for recebido pelo programa, ele será enviado para a função.

A função *olsr_register_timeout_function* registra a função de *timeout*, sendo ela chamada por padrão dez vezes por segundo. No programa, a própria função calcula o tempo de *timeout* pela hora atual e pela hora de armazenamento do valor na estrutura *ett_entry*.

A função *olsr_register_scheduler_event* registra que a função *olsr_event* deverá ser executada no intervalo EMISSION_INTERVAL (4,5 segundos por padrão). Essa função tem por finalidade enviar os pacotes nesse intervalo, conforme já explicado.

O *plugin* permite ainda adicionar uma conexão externa meramente informativa, ou seja, onde qualquer usuário pode se conectar e receber informações sobre o funcionamento do *plugin*. Esse procedimento é definido pelas funções IPC (*InterProcess Communication*), onde um *socket* é registrado. No caso desse programa, o registro se faz na porta 8888. Ele recebe uma saída com as estruturas *ett_compute_time* e *ett_entry*. Abaixo segue um exemplo dessa saída, onde os tempos do campo ETT ENTRY estão representados em nanosegundos:

```
--------------
--ETT ENTRY--
[192.168.0.34]: ETT values: 10.000000, 14.000000, 10.000000, 15.000000,
12.000000, 12.000000, 18.000000, 15.000000, 14.000000, 10.000000,
[192.168.0.2]: ETT values: 549.000000, 894.000000, 542.000000,
71423.000000, 73714.000000, 770.000000, 544.000000, 542.000000,
51628.000000, 546.000000,
[192.168.0.4]: ETT values: 548.000000, 545.000000, 545.000000, 531.000000,
```

```
537.000000, 531.000000, 548.000000, 531.000000, 768.000000, 541.000000,
[192.168.0.10]: ETT values: 538.000000, 544.000000, 538.000000, 536.000000,
66489.000000, 539.000000, 545.000000, 540.000000, 535.000000, 544.000000,
[192.168.0.5]: ETT values: 544.000000, 59482.000000, 542.000000,
535.000000, 544.000000, 544.000000, 535.000000, 541.000000, 56194.000000,
60169.000000,
[192.168.0.1]: ETT values: 491.000000, 491.000000, 53579.000000,
482.000000, 494.000000, 477.000000, 495.000000, 482.000000,
71968738.000000, 490.000000,
[192.168.0.9]: ETT values: 540.000000, 547.000000, 518.000000,
72016369.000000, 541.000000, 518.000000, 532.000000, 532.000000,
71783183.000000, 510.000000,
[192.168.0.33]: ETT values: 12.000000, 12.000000, 13.000000, 13.000000,
12.000000, 13.000000, 11.000000, 12.000000, 12.000000, 13.000000,
--------------
--DIF ENTRY--
[192.168.0.9]: Difference time is:  13.000000usec
[192.168.0.2]: Difference time is:  14.000000usec
[192.168.0.33]: Difference time is:  16.000000usec
[192.168.0.34]: Difference time is:  15.000000usec
[192.168.0.10]: Difference time is:  12.000000usec
[192.168.0.5]: Difference time is:  10.000000usec
--------------
```

Nesse exemplo, DIF ENTRY representa a última diferença de tempo medida pelo nó para cada vizinho, dentre um conjunto de diferenças que ainda serão enviadas. ETT ENTRY representa todas as diferenças recebidas que foram medidas pelos vizinhos de pares de pacotes enviados pelo nó.

4.3. Cenário de Teste

O ambiente de teste foi composto por nove roteadores Linksys cuja descrição foi feita na seção 4.1, de endereços IP: 192.168.0.1, 192.168.0.2, 192.168.0.3, 192.168.0.4, 192.168.0.5, 192.168.0.6, 192.168.0.7, 192.168.0.9 e 192.168.0.10. O ambiente foi composto ainda por quatro microcomputadores:

- Lumiar (192.168.0.31):
    - Processador: AMD Athlon(tm) XP 1250 MHz
    - Memória: 256MB
    - Placa de rede: Atheros Communications, Inc. AR5212 802.11abg
- Buzios (192.168.0.32):

    - Processador: AMD Athlon(tm) XP 1250 MHz
    - Memória: 512MB
    - Placa de rede: Atheros Communications, Inc. AR5212 802.11abg
- Leblon (192.168.0.33):
    - Processador: Intel(R) Pentium(R) 4 CPU 2,80GHz
    - Memória: 1GB
    - Placa de rede: Atheros Communications, Inc. AR5212 802.11abg
- Grajau (192.168.0.34):
    - Processador: Intel(R) Pentium(R) D CPU 3,20GHz
    - Memória: 1GB
    - Placa de rede: Atheros Communications, Inc. AR5212 802.11abg

A distribuição dos equipamentos se deu no laboratório do Grupo de Teleinformática e Automação localizado no Bloco H do Centro de Tecnologia da UFRJ, e nos seus arredores. A interface de rede foi configurada no modo 802.11g, na freqüência de 2,437 GHz (Canal 6), enquanto que no momento do teste havia apenas uma outra rede vizinha operando na freqüência de 2,412 GHz (Canal 1). O teste foi realizado em um fim de semana (06/01/2007), onde há menos atividades e pessoas no prédio que possam influenciar o seu desempenho.

Para os testes foi utilizado o comando `ping` conforme os *scripts* descritos no Apêndice 7.3. Nos testes, 100 pacotes de `ping` (ICMP ECHO REQUEST) são enviados a cada 0,05 segundos. O tamanho dos pacotes varia de 100, 250, 500, 750, 1000, 1250 e 1472 bytes de acordo com a rodada de análise. Esse procedimento é repetido 10 vezes sendo a seguir analisados e montados dois gráficos com o resultado, que mostram a taxa de perda média de pacotes e a quantidade de trocas de rotas para cada tamanho de pacote. O primeiro gráfico contém barras de erro verticais correspondentes a um intervalo de confiança de 95% em torno da média.

A taxa de perda média de pacotes é definida como a sua probabilidade de perda:

$$\text{Probabilidade de perda} = 1 - \left(\frac{\text{Número de pacotes recebidos}}{\text{Número de pacotes enviados}} \right)$$

Enquanto que o número de trocas de rotas representa quantas vezes a rota foi trocada em todas as rodadas de `ping` enviados.

Existem programas voltados para medição de tráfego, como o `iperf` [13], cuja vantagem é calcular medidas como vazão e *jitter* automaticamente. Entretanto, não foi obtido sucesso com a sua utilização porque devido à grande perda de pacotes observada no ambiente de testes, muitas vezes não foi possível iniciar o programa corretamente nos dois nós escolhidos para teste, uma vez que o `ssh` não conectava com sucesso. Uma solução seria executar todo o processo manualmente, o que, entretanto seria inviável em termos de tempo além de incluir o fator de erro humano nesse processo. A vantagem do `ping` é não requerer um programa servidor do outro lado da rede. Porém, as medidas devem ser calculadas separadamente. Além disso, no `ping` se mede taxa de pacotes entregues, enquanto que no `iperf` já é fornecida a vazão.

Foram realizadas medições utilizando três métricas diferentes: número de saltos, ETX e ETT. Foram utilizados sempre dois nós na rede, um microcomputador, grajau (192.168.0.34) e um roteador Linksys (192.168.0.1). Essa escolha foi feita com base na distância física entre esses nós e também porque a baixa capacidade de processamento dos roteadores poderia influenciar nas medições, caso o `ping` fosse efetuado a partir de um dos roteadores.

As métricas foram avaliadas em duas situações: Na primeira medição foram usadas as configurações padrão do OLSR, ou seja, aquelas especificadas no RFC 3626 [5]. No teste com a métrica ETT foi utilizado o mesmo tempo de emissão padrão do pacote HELLO (2

segundos), bem como o mesmo de validade (6 segundos). Na segunda medição foram usadas as seguintes configurações diferentes das padrões, quando aplicável à métrica:

- Janela de medição de 100 amostras (aplicado a ETX e ETT): são utilizadas 100 amostras para se obter a qualidade de um enlace, enquanto que o padrão é de 10 amostras.

- Método FishEye incluído (aplicado a saltos, ETX e ETT): método na qual as mensagens TC são enviadas menos freqüentemente para nós distantes, para isso o TTL dessas mensagens é colocado na seguinte ordem: 255 3 2 1 2 1 1 3 2 1 2 1 1.

- Redundância das mensagens TC (aplicado a saltos, ETX e ETT): ao invés de carregar informações somente sobre os MPRs nas mensagens TC, são enviadas informações acerca de todos os vizinhos.

- Quantidade de MPRs selecionáveis em 7 (aplicado a saltos, ETX e ETT): são selecionados até sete MPRs para atingir um nó vizinho, onde por padrão seria somente um, o que na prática elimina os recursos dos MPRs, mas adicionando redundância no envio de mensagens de controle.

- Intervalo de emissão de mensagens HELLO em 1 segundo (aplicado a saltos, ETX e ETT): diferente do padrão que é de 2 segundos, as mensagens são enviadas em intervalos de 1 segundo. O seu tempo de validade também foi colocado em 100 segundos para compatibilizar com a janela de medição.

- Intervalo de emissão de mensagens TC em 0,5 segundos (aplicado a saltos, ETX e ETT): diferente do padrão que é de 5 segundos, as mensagens são enviadas em intervalos de 0,5 segundos.

- O tempo de emissão do ETT foi colocado em 1 segundo (aplicado a ETT): para compatibilizar com o tempo de emissão de mensagens de HELLO, tanto o tempo de emissão como de expiração foram colocados os mesmos, 1 e 100 segundos.

A Tabela 3 resume os parâmetros usados nos arquivos de configuração, conforme exemplo de arquivo de configuração presente no Apêndice 7.1.3.

Tabela 3: Resumo dos arquivos de configuração.

	HOP não-padrão	**ETX não-padrão**	**ETT não-padrão**	**HOP padrão**	**EXT padrão**	**ETT padrão**
Hysteresis	Não	Não	Não	Não	Não	Não
Janela de HELLO para LQ (amostras)	N/A	100	100	N/A	10	10
LinkQualityFishEye	Sim	Sim	Sim	Não	Não	Não
TcRedundancy	Todos os vizinhos	Todos os vizinhos	Todos os vizinhos	MPRs somente	MPRs somente	MPRs somente
MprCoverage	7	7	7	1	1	1
HelloInterval (segundos)	1	1	1	2	2	2
HelloValidityTime (segundos)	100	100	100	6	6	6
TcInterval (segundos)	0,5	0,5	0,5	5	5	5
TcValidityTime (segundos)	15	15	15	15	15	15
emission_interval (segundos)	N/A	N/A	1	N/A	N/A	2
ett_expiration_time (segundos)	N/A	N/A	100	N/A	N/A	6

4.4. Resultados

Durante os testes, a topologia da rede foi obtida através do *plugin* Dot_draw. Esse *plugin*, através das informações presentes nas mensagens de controle de topologia recebidas, constrói o mapa da topologia, representando cada nó com um círculo ou um retângulo e inserindo o seu endereço IP no interior. Esse mapa é obtido em relação a um nó, no caso o de endereço IP 192.168.0.34. Os nós representados por quadrado são os seus MPRs e os nós representados por círculo são os demais nós. As setas representam a direção com que aquela informação foi recebida pelas mensagens de controle, podendo existir enlaces unidirecionais porque só serão recebidas mensagens dos nós que selecionaram o nó pelo qual o *plugin* está

sendo rodado (192.168.0.34) como MPR. Para as métricas ETX e o ETT, o *plugin* insere ainda informações acerca de seus valores junto à seta de cada salto.

A configuração da rede para a métrica por número de saltos é a definida na Figura 14, enquanto que para a métrica ETX na Figura 15, e para a métrica ETT na Figura 16. Deve-se enfatizar que essa topologia pode variar conforme ocorrem as trocas de mensagem na rede. Note que entre os nós 192.168.0.34 e 192.168.0.1 que são, respectivamente, origem e destino dos testes realizados existem vários caminhos possíveis.

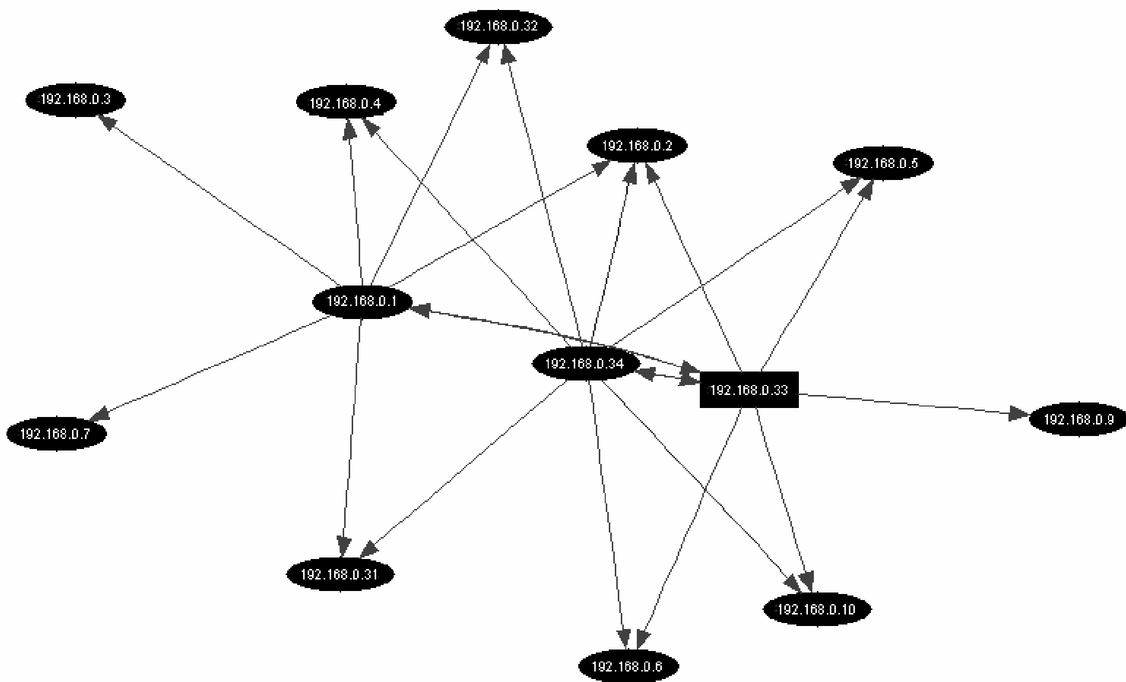


Figura 14: Topologia da métrica contagem de saltos.

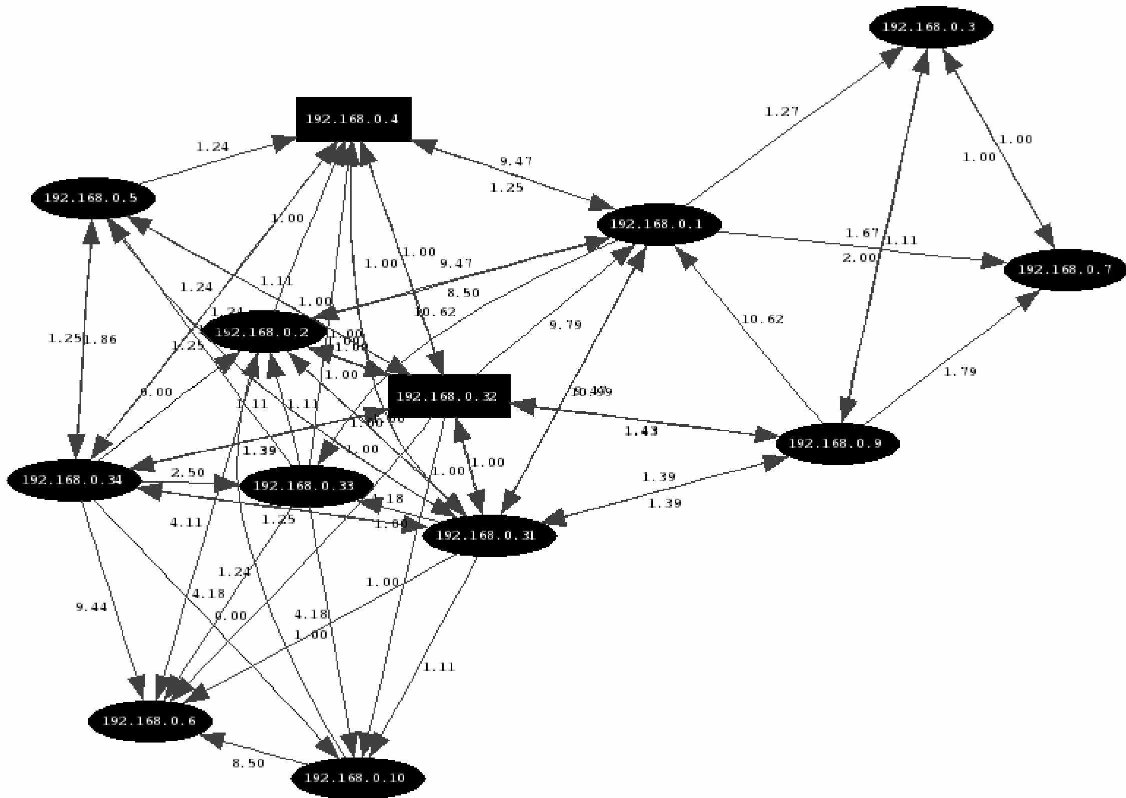


Figura 15: Topologia da métrica ETX.

Figura 16: Topologia da métrica ETT.

O resultado da medida de porcentagem de perdas para a configuração padrão segue na Figura 17 e o número de trocas na Figura 18. Note que a métrica por número de saltos apresenta um desempenho melhor do que as métricas ETT e ETX. Isto ocorre, pois a conectividade foi bem maior, como mostra o gráfico de troca de rotas. Considerando o número de saltos, não houve mudanças de rota durante cada uma das rodadas de teste. Note que não só as perdas foram menores, mas também o número de vezes que a rota variou.

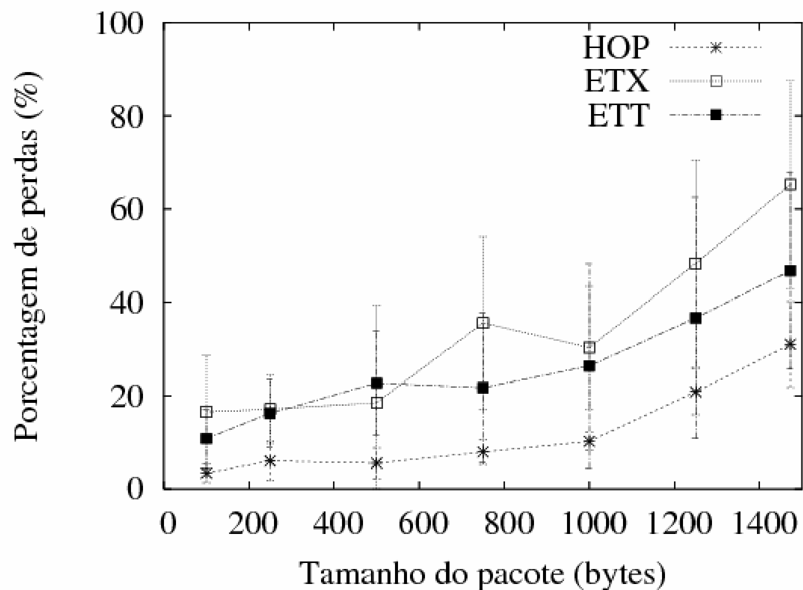


Figura 17: Perda de pacotes para a configuração padrão.

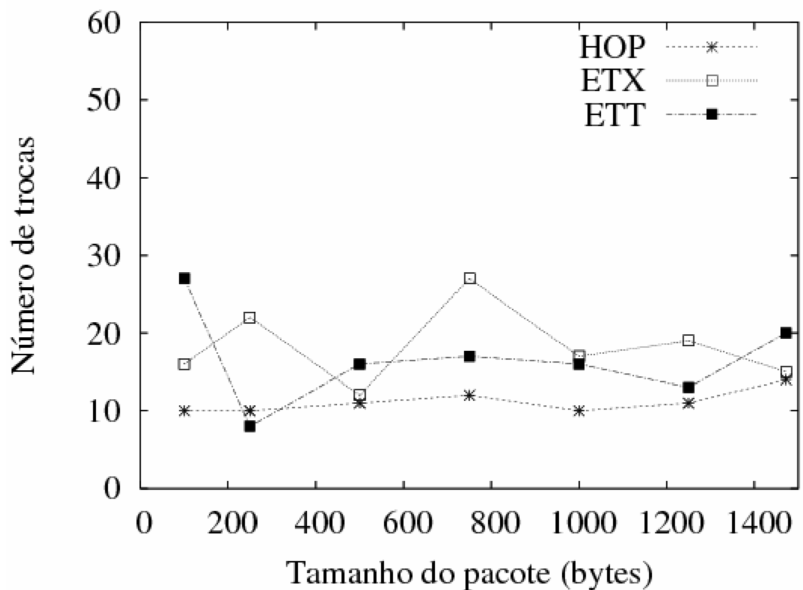


Figura 18: Troca de rota para a configuração padrão.

Para a configuração não-padrão, o resultado da medida de porcentagem de perda segue na Figura 19 e o número de troca de rota na Figura 20. Note que a perda para a métrica por número de saltos foi bem maior em relação ao teste anterior, também apresentando na Figura 19 um desempenho pior que as outras métricas. A métrica ETX apresentou a menor taxa de

erro devido ao fato de se estar usando todos os nós para divulgarem rotas e ao fato de se estar enviando mensagens de HELLO mais freqüentemente. Note que a taxa variação foi bem maior, o que pode ser comprovado pelas barras de erro.

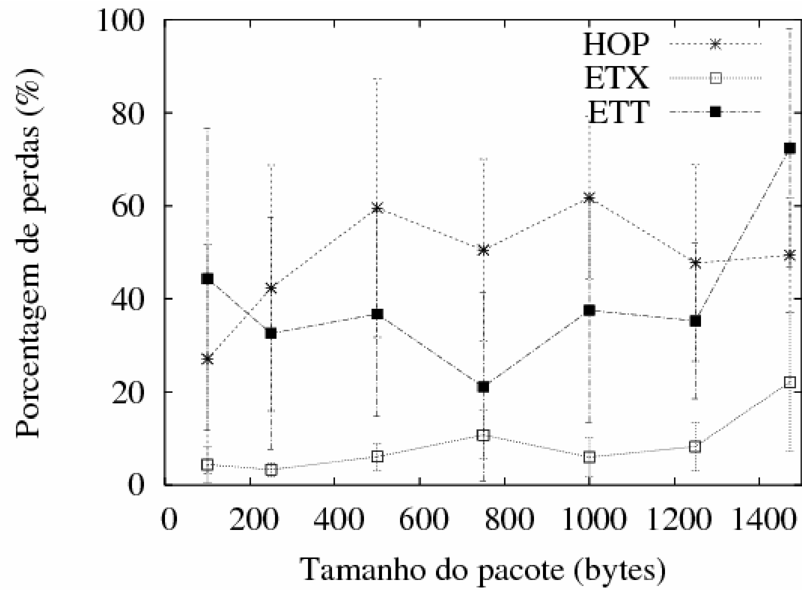


Figura 19: Perda de pacotes para a configuração não-padrão.

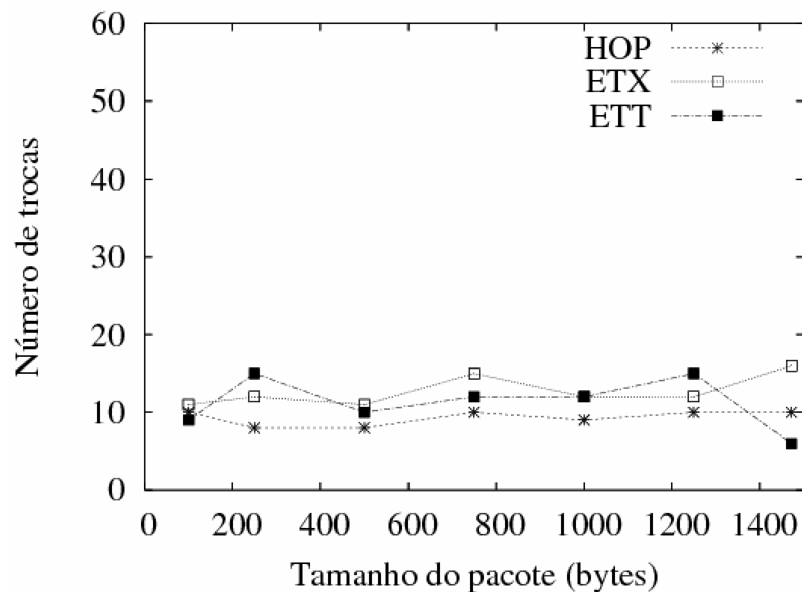


Figura 20: Troca de rota para a configuração não-padrão.

Assim, pode-se verificar que o melhor funcionamento foi com a métrica ETX operando o MPR, pois a quantidade a ser selecionado estava em 7 e não há um nó com mais de 7 vizinhos (vide Figura 15). A métrica ETT apresentou um desempenho em geral sempre pior do que a ETX, comprovando que o seu uso com o OLSR não é vantajoso da maneira como foi implementada.

Como as mensagens de controle do OLSR são enviadas em *broadcast*, as mensagens desse *plugin* também o são. Entretanto essas mensagens são enviadas na taxa básica, 6 Mbits/seg, o que pode influenciar no resultado. Seria mais adequado a essa métrica que os pacotes fossem enviados em *unicast*, o que levaria o problema além da construção de um *plugin*, exigindo mudanças no próprio protocolo.

5. Conclusão

Através desse trabalho foi feito um estudo do protocolo OLSR, incluindo todos os detalhes de seu funcionamento, como por exemplo as trocas de mensagens e o funcionamento dos MPRs, além de uma comparação com outros protocolos existentes. Ainda foi estudada a métrica já existente e usada nesse protocolo, o ETX, para poder compará-la com a métrica alvo desse trabalho, o ETT.

Assim, foi implementado um *plugin* com a métrica ETT para o OLSR. A métrica foi testada em um ambiente de testes real construído com roteadores Linksys e micro-computadores. Deve-se ressaltar a dificuldade da avaliação de uma métrica, pois esta requer ações em tempo real que irão influenciar na qualidade e funcionamento da rede.

Ainda assim, foi obtido um resultado comparando as métricas já utilizadas, contagem de saltos e ETX com a métrica ETT. Foi possível determinar que o ETT, da maneira como foi implementado, não é vantajoso em relação ao ETX.

Seria mais adequado a essa métrica que os pacotes fossem enviados em *unicast*, ou seja, enviar o par de pacotes individualmente para cada vizinho ao invés de enviar para todos como é feito neste trabalho, pois as mensagens de controle do OLSR são enviadas em *broadcast*. A implementação do ETT no OLSR é viável. No entanto, não pode ser feita apenas por um *plugin*, mas exige a modificação do próprio protocolo OLSR. Assim, esta implementação é uma proposta de trabalho futuro.

Também foi possível verificar que para o cenário de testes utilizado, o ETX sem a presença do MPR apresentou um desempenho superior ao mesmo utilizando as configurações padrão. Isso aconteceu porque os nós dispunham de mais informações sobre a rede e mais rotas disponíveis.

Cabe como trabalho futuro, a realização de outros testes comparativos para se obter mais conclusões a respeito das métricas estudadas, como por exemplo, em redes mais espaçadas, com tráfego de dados ou com mobilidade. Cabe ainda utilizar programas como o Iperf para melhores análises das métricas. Pode-se ainda realizar testes acerca das limitações que os roteadores Linksys impõe a rede, bem como a influência que os programas de testes impõe a ele.

6. Referências

[1] Douglas S. J. De Couto, Daniel Aguayo, John Bicket e Robert Morris, "A HighThroughput Path Metric for MultiHop Wireless Routing" - M.I.T. Computer Science and Artificial Intelligence Laboratory.

[2] Richard Draves, Jitendra Padhye e Brian Zill, "Routing in Multi-Radio, Multi-Hop Wireless Mesh Networks", ACM Mobicom, Setembro/2004.

[3] Tønnesen, A. Hafslund, "Implementing and extending the Optimized Link State Routing Protocol", Master Thesis – UniK, Novembro/2000.

[4] Freie funkbasierte Netzwerke, Disponível em http://freifunk.net/downloads/freifunk-praesentation_v11_060804_03_jpn.pdf, último acesso: 10/12/2006.

[5] T. Clausen, P. Jacquet, A Laoiti, P Minet, P. Muhlethaler e A Qayyum, L Viennot, "Optimized Link State Routing Protocol". IETF RFC 3626 Outubro/2003.

[6] Tønnesen, A. Hafslund e Ø. Kure, "The UniK OLSR plugin Interface". OLSR interop workshop em San Diego, Agosto/2004.

[7] Ian F. Akyildiz, Xudong Wang e Weilin Wang, "Wireless mesh networks: a survey", Computer Networks, Volume 47, Issue 4, Março 2005, Páginas 445 a 487.

[8] Johnson, Maltz e Hu, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)". *Internet-Draft* , Abril/2003. Disponível em http://www.cs.cmu.edu/~dmaltz/internet-drafts/draft-ietf-manet-dsr-09.txt.

[9] S.R. Das, C.E. Perkins e E.M. Royer, "Performance Comparison of Two On-Demand Routing Protocols for Ad Hoc Networks", IEEE Personal Communications, Fevereiro/2001.

[10] G. Malkin, "RIP Version 2". Standard, Novembro 1998. Disponível em http://tools.ietf.org/html/rfc2453.

[11] Zygmunt J. Haas, Marc R. Pearlman e Prince Samar, “The Zone Routing Protocol (ZRP) for Ad Hoc Networks”. *Internet-Draft*, Julho/2002. Disponível em http://www.ietf.org/proceedings/02nov/I-D/draft-ietf-manet-zone-zrp-04.txt.

[12] Endereço: http://www.iana.org/. Último acesso: 05/01/2007.

[13] Endereço: http://dast.nlanr.net/Projects/Iperf/. Último acesso: 05/01/2007.

[14] Perkins C. E., Belding-Royer E. M. e Das S. R., “Ad Hoc On-Demand Distance Vector Routing”, IETF RFC 3561, julho/2003.

[15] Perkins CE e Bhagwat P., “Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers”. Computer Communication Review, 1994; Páginas 234 a 244.

[16] Thomas H. Cormen, Charles E. Leiserson e Ronald L. Rivest, “Introduction to Algorithms”, MIT Press, 1990.

[17] Endereço: http://www.nortel.com/corporate/news/newsreleases/2004d/11_17_04_taipei_city.html. Último acesso: 01/01/2007.

[18] Endereço: http://www.infoworld.com/article/05/10/19/HNgooglewifi_1.html. Último acesso: 01/01/2007.

7. Apêndice

7.1. Código Fonte do *Plugin*

7.1.1. olsrd_plugin.c

```
/*
 * Universidade Federal do Rio de Janeiro
 * Engenharia Eletrônica e de Computação
 * Felipe Ortigão Sampaio Buarque Schiller
 * Projeto Final de Curso - 2006/2
 */

/*
 * Dynamic linked library plugin to implement ETT
 */


#include "olsrd_plugin.h"
#include "olsrd_ett.h"
#include <stdio.h>

void __attribute__ ((constructor))
my_init(void);

void __attribute__ ((destructor))
my_fini(void);


/*
 * Returns the version of the plugin interface that is used
 */
int
olsrd_plugin_interface_version()
{
  return OLSRD_PLUGIN_INTERFACE_VERSION;
}

int
olsrd_plugin_register_param(char *key, char *value)
{
  return 1;
}

/**
 *Constructor
 */
void
my_init()
{
  /* Print plugin info to stdout */
  printf("%s\n", MOD_DESC);
}

/**
 *Destructor
```

```
  */
void
my_fini()
{

  /* Calls the destruction function
   * olsr_plugin_exit()
   */
  olsr_plugin_exit();
}
```

7.1.2. olsrd_ett.h

```
/*
 * Universidade Federal do Rio de Janeiro
 * Engenharia Eletrônica e de Computação
 * Felipe Ortigão Sampaio Buarque Schiller
 * Projeto Final de Curso - 2006/2
 */

/*
 * Dynamic linked library plugin to implement ETT
 */

#ifndef _OLSRD_ETT_PLUGIN
#define _OLSRD_ETT_PLUGIN

#include "defs.h"
#include "olsrd_plugin.h"

#define IPC_PORT           8888

#define EMISSION_INTERVAL  4.5
#define ETT_EXPIRATION_TIME 20
#define SAMPLES_MAX 10

/*Parametric values so that metric modifier must be between 0 and 1 */
#define PARAMETRIC_MOD_MIN 0
#define PARAMETRIC_MOD_MAX 60

#define BIG_PKT_SIZE 1137
#define SMALL_PKT_SIZE 137

/* Database entry */

struct ett_entry /* Fill this struture upon receive probe with data I sent
*/
{
        union olsr_ip_addr     neighbour;  /* IP address of the neighbour
that sent the probe */
        struct timeval         timer;      /* Validity time */
        double                             ett_time[SAMPLES_MAX];    /* List of
time difference to the neighbour */
        short unsigned         last_used;      /* Last sample used */
        struct ett_entry       *next,*prev;
};

struct ett_compute_time /* Fill this struture upon receive probe with time
```

```
each neighbour sent me wants to know */
{
        union olsr_ip_addr     originator;  /* IP address of the neighbour
*/
        struct timeval         big_probe_time, small_probe_time;   /* Time
between probes is the difference (last par) */
        struct ett_compute_time        *next,*prev;
};



/***************************************************************************
***
 *                              Plugin data
*
 
****************************************************************************
**/

#define PLUGIN_NAME    "OLSRD ETT Metric plugin"
#define PLUGIN_VERSION "0.1"
#define PLUGIN_AUTHOR   "Felipe Schiller"
#define MOD_DESC PLUGIN_NAME " " PLUGIN_VERSION " by " PLUGIN_AUTHOR

/* The type of message you will use */
#define BIG_MESSAGE_TYPE 140
#define SMALL_MESSAGE_TYPE 141

/* The type of messages we will receive */
#define BIG_PARSER_TYPE BIG_MESSAGE_TYPE
#define SMALL_PARSER_TYPE SMALL_MESSAGE_TYPE


/***************************************************************************
**
 *                              PACKET SECTION
*
 
****************************************************************************
*/



/*
 * Packet message definition
 */

struct ettmsg
{
  olsr_u32_t           destination[83];       //sizeof(olsr_u32_t) = 4;
1000/4/3 = 83
  double                       ett_time[83];          //sizeof(double) = 8;
1000/8/(2*3) = 83
};

/*
 * OLSR messages
 */

struct ett_big_olsrmsg
{
  olsr_u8_t     olsr_msgtype;
```

```
  olsr_u8_t     olsr_vtime;
  olsr_u16_t    olsr_msgsize;
  olsr_u32_t    originator;
  olsr_u8_t     ttl;
  olsr_u8_t     hopcnt;
  olsr_u16_t    seqno;

  struct ettmsg msg;
  char                 getspace[BIG_PKT_SIZE-1012];   //Rest of the packet
= 12 + 1000 bytes;1137 - 1012 = 125

};

struct ett_small_olsrmsg
{
  olsr_u8_t     olsr_msgtype;
  olsr_u8_t     olsr_vtime;
  olsr_u16_t    olsr_msgsize;
  olsr_u32_t    originator;
  olsr_u8_t     ttl;
  olsr_u8_t     hopcnt;
  olsr_u16_t    seqno;

  char                 getspace[SMALL_PKT_SIZE-12];   //Rest of the packet
= 12 bytes;137 - 12 = 125

};


/* Timeout function to register with the scheduler */
void
olsr_timeout(void);

/* Parser function to register with the scheduler */
void
olsr_parser_big_probe(union olsr_message *, struct interface *, union
olsr_ip_addr *);

void
olsr_parser_small_probe(union olsr_message *, struct interface *, union
olsr_ip_addr *);

/* Event function to register with the scheduler */
void
olsr_event(void *);

void
ipc_action(int fd);

int
update_ett_entry(union olsr_ip_addr *, struct ettmsg *, double,struct
interface *);

void
print_ett_table(void);

void
print_dif_table(void);

int
get_time_difference(struct ettmsg *);
```

```
int
olsrd_plugin_init(void);

int
plugin_ipc_init(void);

void
olsr_plugin_exit(void);

void
olsr_get_timestamp(olsr_u32_t, struct timeval *);

void
olsr_init_timer(olsr_u32_t, struct timeval *);

int
olsr_timed_out(struct timeval *);

void
change_metric(union olsr_ip_addr *,double *, struct interface *);

int
update_ett_compute_time(union olsr_ip_addr *, struct timeval *,int,struct
interface *);

#endif
```

7.1.2. olsrd_ett.c

```
/*
 * Universidade Federal do Rio de Janeiro
 * Engenharia Eletrônica e de Computação
 * Felipe Ortigão Sampaio Buarque Schiller
 * Projeto Final de Curso - 2006/2
 */

/*
 * Dynamic linked library plugin to implement ETT
 */

#include "olsrd_ett.h"
#include "olsrd_plugin.h"

#include "olsr.h"
#include "mantissa.h"
#include "parser.h"
#include "scheduler.h"
#include "link_set.h"
#include "socket_parser.h"
#include "interfaces.h"
#include "duplicate_set.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
```

```
#ifdef OS
#undef OS
#endif

#ifdef WIN32
#define close(x) closesocket(x)
#define OS "Windows"
#endif
#ifdef linux
#define OS "GNU/Linux"
#endif
#ifdef __FreeBSD__
#define OS "FreeBSD"
#endif

#ifndef OS
#define OS "Undefined"
#endif

/* The database */
static struct ett_entry list_ett;
static struct ett_compute_time list_dif;


/* set buffer to size of the message */
static char big_buffer[sizeof(struct ett_big_olsrmsg)];
static char small_buffer[sizeof(struct ett_small_olsrmsg)];

int ipc_socket;
int ipc_open;
int ipc_connection;
int ipc_connected;

int
ipc_send(char *, int);


/**
 *Do initialization here
 */
int
olsrd_plugin_init()
{
  if(olsr_cnf->ip_version != AF_INET)
    {
      fprintf(stderr, "This plugin only supports IPv4!\n");
      return 0;
    }
  /* Initial IPC value */
  ipc_open = 0;

  /* Init list */

  list_ett.next = &list_ett;
  list_ett.prev = &list_ett;

  list_dif.next = &list_dif;
  list_dif.prev = &list_dif;


  /* Register functions with olsrd */
```

```
  olsr_parser_add_function(&olsr_parser_big_probe, BIG_PARSER_TYPE, 1);
  olsr_parser_add_function(&olsr_parser_small_probe, SMALL_PARSER_TYPE, 1);

  olsr_register_timeout_function(&olsr_timeout);

  olsr_register_scheduler_event(&olsr_event, NULL, EMISSION_INTERVAL, 0,
NULL);

  return 1;
}

int
plugin_ipc_init()
{
  struct sockaddr_in sin;
  olsr_u32_t yes = 1;

  /* Init ipc socket */
  if ((ipc_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
      perror("IPC socket");
      return 0;
    }
  else
    {
      if (setsockopt(ipc_socket, SOL_SOCKET, SO_REUSEADDR, (char *)&yes,
sizeof(yes)) < 0)
      {
              perror("SO_REUSEADDR failed");
              return 0;
      }

#if defined __FreeBSD__ && defined SO_NOSIGPIPE
      if (setsockopt(ipc_socket, SOL_SOCKET, SO_NOSIGPIPE, (char *)&yes,
sizeof(yes)) < 0)
      {
              perror("SO_REUSEADDR failed");
              return 0;
      }
#endif

      /* Bind the socket */

      /* complete the socket structure */
      memset(&sin, 0, sizeof(sin));
      sin.sin_family = AF_INET;
      sin.sin_addr.s_addr = INADDR_ANY;
      sin.sin_port = htons(IPC_PORT);

      /* bind the socket to the port number */
      if (bind(ipc_socket, (struct sockaddr *) &sin, sizeof(sin)) == -1)
        {
          perror("IPC bind");
          return 0;
        }

      /* show that we are willing to listen */
      if (listen(ipc_socket, 1) == -1)
        {
          perror("IPC listen");
          return 0;
```

```
        }

      /* Register with olsrd */
      add_olsr_socket(ipc_socket, &ipc_action);

    }

  ipc_open = 1;
  return 1;
}

void
ipc_action(int fd)
{
  struct sockaddr_in pin;
  socklen_t addrlen;
  char *addr;

  addrlen = sizeof(struct sockaddr_in);

  if ((ipc_connection = accept(ipc_socket, (struct sockaddr *)  &pin,
&addrlen)) == -1)
    {
      perror("IPC accept");
      exit(1);
    }
  else
    {
                  addr = inet_ntoa(pin.sin_addr);
                  if(ntohl(pin.sin_addr.s_addr) != INADDR_LOOPBACK)
                {
                  olsr_printf(1, "Front end-connection from foregin host(%s)
not allowed!\n", addr);
                  close(ipc_connection);
                  return;
                }
                  else
                {
                  ipc_connected = 1;
                  olsr_printf(1, "ETT PLUG-IN: Connection from %s\n",addr);
                }
    }

}

/*
 * destructor - called at unload
 */
void
olsr_plugin_exit()
{
  if(ipc_open)
    close(ipc_socket);
}


/**
 *A timeoutfunction called every time
 *the scheduler is polled
 */
void
```

```
olsr_timeout()
{
  struct ett_entry *tmp_list;
  struct ett_entry *entry_to_delete;

      tmp_list = list_ett.next;
      while(tmp_list != &list_ett)
        {
          /*Check if the entry is timed out*/
          if(olsr_timed_out(&tmp_list->timer))
            {
              entry_to_delete = tmp_list;
              tmp_list = tmp_list->next;
              olsr_printf(3, "ETT PLUG-IN: Entry for %s timed out..
deleting it\n",
                          olsr_ip_to_string(&entry_to_delete->neighbour));
              /* Dequeue */
              entry_to_delete->prev->next = entry_to_delete->next;
              entry_to_delete->next->prev = entry_to_delete->prev;

              /* Delete */
              free(entry_to_delete);
            }
          else
              tmp_list = tmp_list->next;
        }

  return;
}


/**
 *Scheduled event
 */
void
olsr_event(void *foo)
{
  struct ett_big_olsrmsg *big_message = (struct
ett_big_olsrmsg*)big_buffer;
  struct ett_small_olsrmsg *small_message = (struct
ett_small_olsrmsg*)small_buffer;
  struct interface *ifn;

  olsr_printf(4, "ETT PLUG-IN: Generating packages - ");

  /* looping trough interfaces */
  for (ifn = ifnet; ifn ; ifn = ifn->int_next)
    {
      olsr_printf(4, "[%s]  \n", ifn->int_name);
      /* Fill message */
      if(olsr_cnf->ip_version == AF_INET)
        {
          /* IPv4 */
          olsr_printf(4, "ETT PLUG-IN: Generating small package\n");

          small_message->olsr_msgtype = SMALL_MESSAGE_TYPE;
          small_message->olsr_vtime = double_to_me(ETT_EXPIRATION_TIME);
          small_message->olsr_msgsize = htons(sizeof(struct
ett_small_olsrmsg));
          memcpy(&small_message->originator, &main_addr, ipsize);
          small_message->ttl = 1;
```

```
          small_message->hopcnt = 0;
          small_message->seqno = htons(get_msg_seqno());

          olsr_printf(4, "ETT PLUG-IN: Generating big package\n");

          big_message->olsr_msgtype = BIG_MESSAGE_TYPE;
          big_message->olsr_vtime = double_to_me(ETT_EXPIRATION_TIME);
          big_message->olsr_msgsize = htons(sizeof(struct
ett_big_olsrmsg));
          memcpy(&big_message->originator, &main_addr, ipsize);
          big_message->ttl = 1;
          big_message->hopcnt = 0;
          big_message->seqno = htons(get_msg_seqno());

          /* Print tables to ipc socket */
      if(!ipc_open)
                plugin_ipc_init();
          print_ett_table();
          print_dif_table();

          /* Get time difference to include in big probe packet */
          get_time_difference(&big_message->msg);

          if(net_outbuffer_push(ifn, (olsr_u8_t *)small_message,
sizeof(struct ett_small_olsrmsg)) != sizeof(struct ett_small_olsrmsg))
            {

              /* Send data and try again */
              net_output(ifn);
              if(net_outbuffer_push(ifn, (olsr_u8_t *)small_message,
sizeof(struct ett_small_olsrmsg)) != sizeof(struct ett_small_olsrmsg))
                olsr_printf(1, "ETT PLUG-IN: could not write small message
to buffer for interface: %s\n", ifn->int_name);
            }
                net_output(ifn);

          if(net_outbuffer_push(ifn, (olsr_u8_t *)big_message,
sizeof(struct ett_big_olsrmsg)) != sizeof(struct ett_big_olsrmsg))
            {

              /* Send data and try again */
              net_output(ifn);
              if(net_outbuffer_push(ifn, (olsr_u8_t *)big_message,
sizeof(struct ett_big_olsrmsg)) != sizeof(struct ett_big_olsrmsg))
                olsr_printf(1, "ETT PLUG-IN: could not write big message to
buffer for interface: %s\n", ifn->int_name);
            }
                net_output(ifn);

        }
      else
        {
            olsr_printf(1,"ETT PLUG-IN: Message is not IPV4!\n");
                return;
        }

    }
  olsr_printf(3, "\n");


  return;
```

```
}


void
olsr_parser_small_probe(union olsr_message *m, struct interface *in_if,
union olsr_ip_addr *in_addr)
{
  struct ett_small_olsrmsg* pm;
  union olsr_ip_addr   originator;
  struct timeval               time_msg_rec;

  gettimeofday(&time_msg_rec,NULL);

  pm = (struct ett_small_olsrmsg*)m;

  /* Fetch the originator of the messsage */
  memcpy(&originator, &m->v4.originator, ipsize);

  /* Fetch the message based on IP version */
  if(olsr_cnf->ip_version != AF_INET)
        {
                olsr_printf(1,"ETT PLUG-IN: Message is not IPV4!\n");
                return;
        }

  /* Check if message originated from this node - impossible to happen */
  if(memcmp(&originator, &main_addr, ipsize) == 0)
    /* If so - back off */
    return;

  /* Check that the neighbor this message was received
     from is symmetric - this is only informative */
  if(check_neighbor_link(in_addr) != SYM_LINK)
      olsr_printf(4, "Received ETT from NON SYM neighbor %s\n",
olsr_ip_to_string(in_addr));

  /*
   * Check if this message has been processed before
   */
  if(!olsr_check_dup_table_proc(&originator, ntohs(m->v4.seqno)))
    {
      /* If so - do not process */
                return;
    }

  /* Process */

  olsr_printf(4, "ETT PLUG-IN: Processing small ETT probe from %s seqno:
%d\n",
              olsr_ip_to_string(&originator),
              ntohs(m->v4.seqno));

  /* Call a function that updates the database entry */
  update_ett_compute_time(&originator,
&time_msg_rec,SMALL_PARSER_TYPE,in_if);
}



void
olsr_parser_big_probe(union olsr_message *m, struct interface *in_if, union
```

```
olsr_ip_addr *in_addr)
{
  struct ett_big_olsrmsg* pm;
  struct ettmsg                *message;
  union olsr_ip_addr   originator;
  double                              vtime;
  struct timeval               time_msg_rec;

  gettimeofday(&time_msg_rec,NULL);

  pm = (struct ett_big_olsrmsg*)m;

  /* Fetch the originator of the messsage */
  memcpy(&originator, &m->v4.originator, ipsize);

  /* Fetch the message based on IP version */
  if(olsr_cnf->ip_version == AF_INET)
    {
      message = &pm->msg;
      vtime = ME_TO_DOUBLE(m->v4.olsr_vtime);
    }
       else
       {
               olsr_printf(1,"ETT PLUG-IN: Message is not IPV4!\n");
               return;
       }

  /* Check if message originated from this node - impossible to happen */
  if(memcmp(&originator, &main_addr, ipsize) == 0)
    /* If so - back off */
    return;

  /* Check that the neighbor this message was received
     from is symmetric */
  if(check_neighbor_link(in_addr) != SYM_LINK)
      olsr_printf(4, "Received ETT from NON SYM neighbor %s\n",
olsr_ip_to_string(in_addr));

  /*
   * Check if this message has been processed before
   */
  if(!olsr_check_dup_table_proc(&originator, ntohs(m->v4.seqno)))
    {
      /* If so - do not process */
               return;
    }

  /* Process */

  olsr_printf(4, "ETT PLUG-IN: Processing big ETT probe from %s seqno:
%d\n",
              olsr_ip_to_string(&originator),
              ntohs(m->v4.seqno));

  /* Call a function that updates the database entry */
  update_ett_compute_time(&originator,
&time_msg_rec,BIG_PARSER_TYPE,in_if);
  update_ett_entry(&originator, message, vtime,in_if);

}
```

```
/**
 *Update or register a new ett entry and calculate metric
 */
int
update_ett_compute_time(union olsr_ip_addr *originator, struct timeval
*time_rec,int message_type ,struct interface *in_if)
{
  struct ett_compute_time *entry;
//  double      time_now;

//  time_now = time_rec->tv_sec*1000000+time_rec->tv_usec;
//  olsr_printf(1, "ETT PLUG-IN: New ett big time from %s is: %fusec\n",
//        olsr_ip_to_string(originator),time_now);

  /* Check for the entry */
  for(entry = list_dif.next;
      entry != &list_dif;
      entry = entry->next)
    {
                  if(memcmp(originator, &entry->originator, ipsize) == 0)
//If found, I haven't transmited...
                {
                          if (message_type == BIG_PARSER_TYPE)
                                entry->big_probe_time = *time_rec;
                          else //(message_type == SMALL_PARSER_TYPE)
                          {
                                entry->small_probe_time = *time_rec;
                                entry->big_probe_time.tv_sec = 0;
                                entry->big_probe_time.tv_usec = 0;
                          }
                  return 0;
                }
    }

  olsr_printf(1, "ETT PLUG-IN: Adding entry to list\n");

  if (message_type == SMALL_PARSER_TYPE)
  {
          entry = olsr_malloc(sizeof(struct ett_compute_time), "ETT PLUG-
IN: new ett entry");

          /* Fill struct */

          memcpy(&entry->originator, originator, ipsize);
          entry->big_probe_time.tv_sec = 0;
          entry->big_probe_time.tv_usec = 0;
      entry->small_probe_time = *time_rec;

          /* Queue */
          entry->next = list_dif.next;
          entry->prev = &list_dif;
          list_dif.next->prev = entry;
          list_dif.next = entry;
  }

  return 1;
}
```

```
/**
 *Update or register a new ett entry and calculate metric
 */
int
update_ett_entry(union olsr_ip_addr *originator, struct ettmsg *message,
double vtime,struct interface *in_if)
{
  struct ett_entry *entry;
  unsigned i=0;

  /* Search packets going to me */
  for(i=0; /*message->destination[i] != 0 && */i < sizeof(message-
>destination); i++)
          if (memcmp(&main_addr,&message->destination[i],ipsize) == 0)
                break;
  if (message->destination[i] == 0) return 0; //No information about my
node

  olsr_printf(1, "ETT PLUG-IN: New ett time difference from %s is: %f\n",
          olsr_ip_to_string(originator),message->ett_time[i]);

  /* Check for the entry */
  for(entry = list_ett.next;
      entry != &list_ett;
      entry = entry->next)
    {
                  if(memcmp(originator, &entry->neighbour, ipsize) == 0)
                {
                  if (entry->last_used == SAMPLES_MAX - 1)
                          entry->last_used = 0;
                  else
                          entry->last_used++;
                  entry->ett_time[entry->last_used] = message->ett_time[i];
                  olsr_get_timestamp(vtime * 1000, &entry->timer);

                  /* Change Metric */
                  change_metric(originator,entry->ett_time,in_if);

                  return 0;
                }
    }

  olsr_printf(1, "ETT PLUG-IN: Adding entry to list\n");


  entry = olsr_malloc(sizeof(struct ett_entry), "ETT PLUG-IN: new ett
entry");

  /* Fill struct */

  memcpy(&entry->neighbour, originator, ipsize);
  entry->ett_time[0] = message->ett_time[i];
  entry->last_used = 0;
  olsr_get_timestamp(vtime * 1000, &entry->timer);

  for (i=1; i < SAMPLES_MAX; i++)
          entry->ett_time[i] = -1;

  /* Queue */
  entry->next = list_ett.next;
  entry->prev = &list_ett;
```

```
  list_ett.next->prev = entry;
  list_ett.next = entry;

  return 1;
}



void change_metric(union olsr_ip_addr *originator,double *ett_time, struct
interface *in_if)
{
  unsigned short       i=0;
  double                      metric_modifier;
  struct link_entry *link;
  struct olsr_if       *cfg_inter;
  struct olsr_lq_mult *mult, *new_mult, *prev_mult;

  /* find the interface configuration for the interface */
  for (cfg_inter = olsr_cnf->interfaces; cfg_inter != NULL;
       cfg_inter = cfg_inter->next)
    if (cfg_inter->interf == in_if)
      break;

  /* Calculate value transformation */
  metric_modifier = (BIG_PKT_SIZE/ett_time[0]-
PARAMETRIC_MOD_MIN)/PARAMETRIC_MOD_MAX;
  for (i=0;i < SAMPLES_MAX; i++)
  {
          if (ett_time[i] == -1)
          {
                  olsr_printf(3, "ETT PLUG-IN: I dont have %i samples for
%s, I only have %i\n",

        SAMPLES_MAX,olsr_ip_to_string(originator),i);
                  return;
          }
          if ((BIG_PKT_SIZE/ett_time[i]-
PARAMETRIC_MOD_MIN)/PARAMETRIC_MOD_MAX > metric_modifier)
                metric_modifier = (BIG_PKT_SIZE/ett_time[i]-
PARAMETRIC_MOD_MIN)/PARAMETRIC_MOD_MAX;
  }

  //Limit range anyway
  if (metric_modifier <= 0)
          metric_modifier = 0.001;
  if (metric_modifier > 1)
          metric_modifier = 1;

  /* loop through the multiplier entries */

  prev_mult = NULL;
  for (mult = cfg_inter->cnf->lq_mult; mult != NULL; mult = mult->next)
  {

    /* Search if an entry already exists */

    if (COMP_IP(&mult->addr, originator))
          {
                mult->val = metric_modifier;
                break;
          }
```

```
        prev_mult = mult;
  }
  if (mult == NULL)
  {
            new_mult = olsr_malloc(sizeof(struct olsr_lq_mult), "ETT PLUG-
IN: new mult entry");
                memcpy(&new_mult->addr, originator, sizeof(union
olsr_ip_addr));
                new_mult->val = metric_modifier;
                new_mult->next=NULL;
                if (cfg_inter->cnf->lq_mult == NULL)
                        cfg_inter->cnf->lq_mult = new_mult;
                else
                        prev_mult->next = new_mult;
  }

  link = get_link_set();

   if (link==NULL)
        olsr_printf(1, "ETT PLUG-IN: Ja recebi link NULL!\n");

  /* Walk through this structure searching the corresponding interface to
insert metric difference */
  while(link)
    {
      if(COMP_IP(originator, &link->neighbor_iface_addr) &&
                        COMP_IP(&main_addr, &link->local_iface_addr))
                 break;
      link = link->next;
    }

  if (link==NULL)
        olsr_printf(1, "ETT PLUG-IN: Error when searching in link_set
struct!\n");
  else
        {
                link->loss_link_multiplier = metric_modifier;
                olsr_printf(3, "ETT PLUG-IN: link->loss_link_multiplier =
%f\n",link->loss_link_multiplier);
        }

  return;
}




/**
 *Print all registered ett entries
 */
void
print_ett_table()
{
  struct ett_entry *entry;
  char buf[200];
  unsigned i;

  if(!ipc_connection)
    return;

  ipc_send("--ETT ENTRY--\n", 14);
```

```
        /* Check for the entry */
        for(entry = list_ett.next;
        entry != &list_ett;
        entry = entry->next)
          {
                sprintf(buf, "[%s]: ", olsr_ip_to_string(&entry-
>neighbour));
                ipc_send(buf, strlen(buf));
                sprintf(buf,"ETT values: ");
                ipc_send(buf, strlen(buf));
                for (i=0; i < SAMPLES_MAX; i++)
                {
                        sprintf(buf,"%f, ",entry->ett_time[i]);
                        ipc_send(buf, strlen(buf));
                }
                ipc_send("\n",1);
          }

        ipc_send("--------------\n", 15);

}

/**
 *Print all registered time difference entries
 */
void
print_dif_table()
{
  struct ett_compute_time *entry;
  char buf[200];
  double time_difference;
  struct timeval diff;

  if(!ipc_connection)
    return;

  ipc_send("--DIF ENTRY--\n", 14);

        /* Check for the entry */
        for(entry = list_dif.next;
        entry != &list_dif;
        entry = entry->next)
          {
                sprintf(buf, "[%s]: ", olsr_ip_to_string(&entry-
>originator));
                ipc_send(buf, strlen(buf));
                sprintf(buf,"Difference time is: ");
                ipc_send(buf, strlen(buf));

                if (timercmp(&entry->big_probe_time,&entry-
>small_probe_time,<))
                        sprintf(buf," Erro: Big probe recebido primeiro que
small probe\n");
                else
                {
                        timersub(&entry->big_probe_time,&entry-
>small_probe_time,&diff);
                        time_difference =
diff.tv_sec*1000000+(diff.tv_usec);
                        sprintf(buf," %fusec\n",time_difference);
```

```
                }
                ipc_send(buf, strlen(buf));
            }

        ipc_send("---------------\n", 15);

}


int
ipc_send(char *data, int size)
{
  if(!ipc_connected)
    return 0;

#if defined __FreeBSD__ || defined __NetBSD__ || defined __OpenBSD__ ||
defined __MacOSX__
  if (send(ipc_connection, data, size, 0) < 0)
#else
  if (send(ipc_connection, data, size, MSG_NOSIGNAL) < 0)
#endif
    {
      olsr_printf(1, "(OUTPUT)IPC connection lost!\n");
      close(ipc_connection);
      ipc_connected = 0;
      return -1;
    }

  return 1;
}

/**
 *Get time difference between probes and delete the structure
 */
int
get_time_difference(struct ettmsg *msg)
{
  struct ett_compute_time *tmp, *entry_to_delete;
  struct timeval diff;
  unsigned i=1;//MIPS workaround
  unsigned j=0;

//Limpa estrutura
for (j=0 ;j < sizeof(&msg->destination); j++)
{
        msg->destination[j] = 0;
        msg->ett_time[j] = -1;
}

  tmp = list_dif.next;
  while(tmp != &list_dif)
        {
                  if (!timercmp(&tmp->big_probe_time,&tmp-
>small_probe_time,>))
                  {
                          tmp = tmp->next;
                          continue;
                  }
                  memcpy(&msg->destination[i],&tmp->originator,ipsize);

                  timersub(&tmp->big_probe_time,&tmp-
```

```
>small_probe_time,&diff);

                  msg->ett_time[i] = diff.tv_sec*1000000+diff.tv_usec;
//nanoseconds
                  i++;

                  /* Deleting entry */
                  entry_to_delete = tmp;
                  tmp = tmp->next;
                  entry_to_delete->prev->next = entry_to_delete->next;
                  entry_to_delete->next->prev = entry_to_delete->prev;
                  free(entry_to_delete);
        }

  return 1;
}




/*************************************************************
 *                  TOOLS DERIVED FROM OLSRD                 *
 *************************************************************/


/**
 *Checks if a timer has times out. That means
 *if it is smaller than present time.
 *@param timer the timeval struct to evaluate
 *@return positive if the timer has not timed out,
 *0 if it matches with present time and negative
 *if it is timed out.
 */
int
olsr_timed_out(struct timeval *timer)
{
  return(timercmp(timer, &now, <));
}



/**
 *Initiates a "timer", wich is a timeval structure,
 *with the value given in time_value.
 *@param time_value the value to initialize the timer with
 *@param hold_timer the timer itself
 *@return nada
 */
void
olsr_init_timer(olsr_u32_t time_value, struct timeval *hold_timer)
{
  olsr_u16_t  time_value_sec;
  olsr_u16_t  time_value_msec;

  time_value_sec = time_value/1000;
  time_value_msec = time_value-(time_value_sec*1000);

  hold_timer->tv_sec = time_value_sec;
  hold_timer->tv_usec = time_value_msec*1000;
}
```

```
/**
 *Generaties a timestamp a certain number of milliseconds
 *into the future.
 *
 *@param time_value how many milliseconds from now
 *@param hold_timer the timer itself
 *@return nada
 */
void
olsr_get_timestamp(olsr_u32_t delay, struct timeval *hold_timer)
{
  olsr_u16_t  time_value_sec;
  olsr_u16_t  time_value_msec;

  time_value_sec = delay/1000;
  time_value_msec= delay - (delay*1000);

  hold_timer->tv_sec = now.tv_sec + time_value_sec;
  hold_timer->tv_usec = now.tv_usec + (time_value_msec*1000);
}
```

7.1.3. Olsrd_ett_rot.conf

```
#
# olsr.org OLSR daemon config file
#
# Lines starting with a # are discarded
#
# This file was shipped with olsrd 0.X.X
#

# Debug level(0-9)
# If set to 0 the daemon runs in the background

DebugLevel      0

# IP version to use (4 or 6)

IpVersion       4

# Clear the screen each time the internal state changes

ClearScreen     yes

# HNA IPv4 routes
# syntax: netaddr netmask
# Example Internet gateway:
# 0.0.0.0 0.0.0.0

Hna4
```

```
{
#   Internet gateway:
#   0.0.0.0      0.0.0.0
#   more entries can be added:
#   192.168.1.0  255.255.255.0
}

# HNA IPv6 routes
# syntax: netaddr prefix
# Example Internet gateway:
Hna6
{
#   Internet gateway:
#   ::             0
#   more entries can be added:
#   fec0:2200:106:: 48
}


# Should olsrd keep on running even if there are
# no interfaces available? This is a good idea
# for a PCMCIA/USB hotswap environment.
# "yes" OR "no"

AllowNoInt     yes

# TOS(type of service) value for
# the IP header of control traffic.
# If not set it will default to 16

#TosValue      16

# The fixed willingness to use(0-7)
# If not set willingness will be calculated
# dynamically based on battery/power status
# if such information is available

#Willingness              4

# Allow processes like the GUI front-end
# to connect to the daemon.

IpcConnect
{
    # Determines how many simultaneously
    # IPC connections that will be allowed
    # Setting this to 0 disables IPC

    MaxConnections  30
```

```
    # By default only 127.0.0.1 is allowed
    # to connect. Here allowed hosts can
    # be added

    Host        127.0.0.1
    #Host        10.0.0.5

    # You can also specify entire net-ranges
    # that are allowed to connect. Multiple
    # entries are allowed

    Net         192.168.0.0 255.255.255.0
}

# Whether to use hysteresis or not
# Hysteresis adds more robustness to the
# link sensing but delays neighbor registration.
# Used by default. 'yes' or 'no'

UseHysteresis no

# Hysteresis parameters
# Do not alter these unless you know
# what you are doing!
# Set to auto by default. Allowed
# values are floating point values
# in the interval 0,1
# THR_LOW must always be lower than
# THR_HIGH.

HystScaling   0.05
HystThrHigh   0.30
HystThrLow    0.001


# Link quality level
# 0 = do not use link quality
# 1 = use link quality for MPR selection
# 2 = use link quality for MPR selection and routing
# Defaults to 0

LinkQualityLevel      2

# Link quality window size
# Defaults to 10

LinkQualityWinSize   100

# Polling rate in seconds(float).
# Default value 0.05 sec
```

```
Pollrate        0.05

LinkQualityFishEye 1

# TC redundancy
# Specifies how much neighbor info should
# be sent in TC messages
# Possible values are:
# 0 - only send MPR selectors
# 1 - send MPR selectors and MPRs
# 2 - send all neighbors
#
# defaults to 0

TcRedundancy        2

#
# MPR coverage
# Specifies how many MPRs a node should
# try select to reach every 2 hop neighbor
#
# Can be set to any integer >0
#
# defaults to 1

MprCoverage  7

LoadPlugin "olsrd_ett.so.0.1"
{
        PlParam "emission_interval" "1.0"
        PlParam "ett_expiration_time" "100.0"
}

# Olsrd plugins to load
# This must be the absolute path to the file
# or the loader will use the following scheme:
# - Try the paths in the LD_LIBRARY_PATH
#   environment variable.
# - The list of libraries cached in /etc/ld.so.cache
# - /lib, followed by /usr/lib

# Example plugin entry with parameters:

#LoadPlugin "olsrd_dyn_gw.so.0.3"
#{
    # Here parameters are set to be sent to the
    # plugin. Theese are on the form "key" "value".
    # Parameters ofcause, differs from plugin to plugin.
    # Consult the documentation of your plugin for details.
```

```
    # Example: dyn_gw params

    # how often to check for Internet connectivity
    # defaults to 5 secs
#   PlParam     "Interval"   "40"

    # if one or more IPv4 addresses are given, do a ping on these in
    # descending order to validate that there is not only an entry in
    # routing table, but also a real internet connection. If any of
    # these addresses could be pinged successfully, the test was
    # succesful, i.e. if the ping on the 1st address was successful,the
    # 2nd won't be pinged
#   PlParam     "Ping"       "141.1.1.1"
#   PlParam     "Ping"       "194.25.2.129"
#}



# Interfaces and their rules
# Omitted options will be set to the
# default values. Multiple interfaces
# can be specified in the same block
# and multiple blocks can be set.

# !!CHANGE THE INTERFACE LABEL(s) TO MATCH YOUR INTERFACE(s)!!
# (eg. wlan0 or eth1):

Interface "eth1"
{

    # IPv4 broadcast address to use. The
    # one usefull example would be 255.255.255.255
    # If not defined the broadcastaddress
    # every card is configured with is used

    # Ip4Broadcast                255.255.255.255

    # IPv6 address scope to use.
    # Must be 'site-local' or 'global'

    # Ip6AddrType                 site-local

    # IPv6 multicast address to use when
    # using site-local addresses.
    # If not defined, ff05::15 is used

    # Ip6MulticastSite            ff05::11

    # IPv6 multicast address to use when
```

```
    # using global addresses
    # If not defined, ff0e::1 is used

    # Ip6MulticastGlobal        ff0e::1


    # Emission intervals.
    # If not defined, RFC proposed values will
    # be used in most cases.

    # Hello interval in seconds(float)
     HelloInterval    1.0

    # HELLO validity time
     HelloValidityTime 100.0

    # TC interval in seconds(float)
     TcInterval       0.5

    # TC validity time
     TcValidityTime     15.0

    # MID interval in seconds(float)
    # MidInterval          5.0

    # MID validity time
    # MidValidityTime 15.0

    # HNA interval in seconds(float)
    # HnaInterval          5.0

    # HNA validity time
    # HnaValidityTime            15.0

    # When multiple links exist between hosts
    # the weight of interface is used to determine
    # the link to use. Normally the weight is
    # automatically calculated by olsrd based
    # on the characteristics of the interface,
    # but here you can specify a fixed value.
    # Olsrd will choose links with the lowest value.

    # Weight 0


}
```

7.2. Código dos arquivos necessários para gerar o IPK

7.2.1. Makefile

```
include $(TOPDIR)/rules.mk

PKG_NAME:=ettmetric
PKG_VERSION:=0.1
PKG_RELEASE:=1
PKG_MD5SUM:=eb0e357ad90d150d4c5a1850d28ebd82

PKG_SOURCE_URL:=http://gta.ufrj.br/~fschiller/
PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2
PKG_CAT:=bzcat

PKG_BUILD_DIR:=$(BUILD_DIR)/$(PKG_NAME)-$(PKG_VERSION)
PKG_INSTALL_DIR:=$(PKG_BUILD_DIR)/ipkg-install

include $(TOPDIR)/package/rules.mk

$(eval $(call PKG_template,ETTMETRIC,ettmetric,$(PKG_VERSION)-
$(PKG_RELEASE),$(ARCH)))

$(PKG_BUILD_DIR)/.configured: $(PKG_BUILD_DIR)/.prepared
        $(SED) "s,^TOPDIR.*,TOPDIR=../olsrd-0.4.10," \
                $(PKG_BUILD_DIR)/Makefile
        #Since there is no configure script, we can directly go to the
building step
        touch $@

$(PKG_BUILD_DIR)/.built:
        rm -rf $(PKG_INSTALL_DIR)
        mkdir -p $(PKG_INSTALL_DIR)/usr/lib
                #Note here that we pass cross-compiler as default compiler
to use
                $(MAKE) -C $(PKG_BUILD_DIR) \
                        $(TARGET_CONFIGURE_OPTS) \
                        NODEBUG=1 \
                        OFLAGS="$(TARGET_CFLAGS)" \
                        OS="linux" \
                        INSTALL_PREFIX="$(PKG_INSTALL_DIR)" \
                        STRIP="/bin/true"
        $(CP) $(PKG_BUILD_DIR)/olsrd_ett.so.0.1 $(PKG_INSTALL_DIR)/usr/lib
        touch $@

$(IPKG_ETTMETRIC):
        mkdir -p $(IDIR_ETTMETRIC)/usr/lib
        cp $(PKG_INSTALL_DIR)/usr/lib/olsrd_ett.so.0.1
$(IDIR_ETTMETRIC)/usr/lib/
        $(STRIP) $(IDIR_ETTMETRIC)/usr/lib/*
        $(IPKG_BUILD) $(IDIR_ETTMETRIC) $(PACKAGE_DIR)

mostlyclean:
        $(MAKE) -C $(PKG_BUILD_DIR) clean
        rm -f $(PKG_BUILD_DIR)/.built
```

7.2.2. Config.in

```
config BR2_PACKAGE_ETTMETRIC
        tristate "olsrd ett metric routing protocol"
        default m if CONFIG_DEVEL
        select BR2_PACKAGE_OLSRD
        help
                ETT metric for adhoc networks
```

7.2.3. ettmetric.control

```
Package: ettmetric
Priority: optional
Section: net
Description: Routing protocol
```

7.3. Scripts de teste

7.3.1. roda_tamanho_pacote.sh

```
#!/bin/tcsh

############################################
############## Checa argumentos #############
############################################

if ( $1 == "--help" ) then
        echo "Uso:./roda_tamanho_pacote.sh <origem> <destino>
<metrica:HOP/ETX/ETT>"
        exit 1
endif

if ( $# < 3) then
       echo "Uso:./roda_tamanho_pacote.sh <origem> <destino>
<metrica:HOP/ETX/ETT>"
       exit 2
endif

############################################
############## Roda o ping ################
############################################

set ntries = 10 # numero de rodadas

set try = 1 # rodada atual
```

```
set src = 192.168.0.$1 # origem do ping
set dst = 192.168.0.$2 # destino do ping
set pause = 30 # intervalo entre as rodadas
set numPkts = 100 # total de pacotes
set interval = 0.05 # intervalo de envio entre pacotes


set SEC = `date +%S`
set MIN = `date +%M`
set HOUR = `date +%H`
set DAY = `date +%d`
set MONTH = `date +%m`
set YEAR = `date +%Y`
set DATE = $HOUR$MIN$SEC-$DAY$MONTH$YEAR

set DATA_DIR = traces.tam.pacote-$1-$2-$3-$DATE
mkdir $DATA_DIR

while ( $try <= $ntries )
        foreach size (100 250 500 750 1000 1250 1472) # tamanho do pacote

                echo "Rodada $try Tamanho $size"
                ping -s $size -c $numPkts -i $interval -R $dst >>
$DATA_DIR/trace.size.$size
                sleep $pause

         end
        @ try+=1
end



############################################
######### Tratamento dos Dados #############
############################################

echo "Tratamento de dados"

# Seleciona a coluna com a vazï¿½ e o tamanho do pacote, e junta os
arquivos
foreach size (100 250 500 750 1000 1250 1472)

        # Porcentagem de perda de pings
        ./analise_ping_perda.awk -v size=$size \
        ./$DATA_DIR/trace.size.$size | sed 's/%//'  >>
./$DATA_DIR/perda.cada.size.$size

        ./ic_miguel.awk nrvar=1 ic=95 \
        ./$DATA_DIR/perda.cada.size.$size >> ./$DATA_DIR/perda.final

        # Numero medio de vezes que trocou de rota
        ./analise_ping_troca.awk -v size=$size \
        ./$DATA_DIR/trace.size.$size >> ./$DATA_DIR/troca.cada.size.$size

        ./ic_miguel.awk nrvar=1 ic=95 \
        ./$DATA_DIR/troca.cada.size.$size >> ./$DATA_DIR/troca.final
end

############################################
################ Plotar ####################
```

```
############################################

echo "Geracao de graficos"

# Copiar os .gnu para dentro do diretorio de dados
cp plota.perda.gnu $DATA_DIR
cp plota.troca.gnu $DATA_DIR

# Entrar no diretorio de dados
cd $DATA_DIR

# Rodar o gnuplot
gnuplot plota.perda.gnu
gnuplot plota.troca.gnu
```

7.3.2 ic_miguel.awk

```
#!/usr/bin/awk -f
{
  xvalues[x=$1] = $1;
  amostras[x=$1]++;
  for (j=1;j<=nrvar;j++) {
      n=j+1;
      v[$1,amostras[x=$1],j]=$n;
  }
}
END {
  if (ic==90) {
      t[20] = 1.729;
      t[15] = 1.761;
      t[10] = 1.833;
      t[5] = 2.132;
      t[3] = 2.920;
  }
  if (ic==95) {
        t[2] = 12.70620474;
        t[3] = 4.30265273;
        t[4] = 3.18244631;
        t[5] = 2.77644511;
        t[6] = 2.57058184;
        t[7] = 2.44691185;
        t[8] = 2.36462425;
        t[9]= 2.30600414;
        t[10] = 2.26215716;
        t[11] = 2.22813885;
        t[12] = 2.20098516;
        t[13] = 2.17881283;
        t[14] = 2.16036866;
        t[15] = 2.14478669;
        t[16] = 2.13144955;
        t[17] = 2.11990530;
        t[18] = 2.10981558;
        t[19] = 2.10092204;
        t[20] = 2.09302405;
        t[21] = 2.08596345;
        t[22] = 2.07961384;
        t[23] = 2.07387307;
        t[24] = 2.06865761;
        t[25] = 2.06389856;
```

```
        t[26] = 2.05953855;
        t[27] = 2.05552944;
        t[28] = 2.05183052;
        t[29] = 2.04840714;
        t[30] = 2.04522964;
        t[31] = 2.04227246;
        t[32] = 2.03951345;
        t[33] = 2.03693334;
        t[34] = 2.03451530;
        t[35] = 2.03224451;
        t[36] = 2.03010793;
        t[37] = 2.02809400;
        t[38] = 2.02619246;
        t[39] = 2.02439416;
        t[40] = 2.02269092;
        t[41] = 2.02107539;
        t[42] = 2.01954097;
        t[43] = 2.01808170;
        t[44] = 2.01669220;
        t[45] = 2.01536757;
        t[46] = 2.01410339;
        t[47] = 2.01289560;
        t[48] = 2.01174051;
        t[49] = 2.01063476;
        t[50] = 2.00957524;
        t[51] = 2.00855911;
        t[52] = 2.00758377;
        t[53] = 2.00664681;
        t[54] = 2.00574600;
        t[55] = 2.00487929;
        t[56] = 2.00404478;
        t[57] = 2.00324072;
        t[58] = 2.00246546;
        t[59] = 2.00171748;
        t[60] = 2.00099538;
        t[61] = 2.00029782;
        t[62] = 1.99962358;
        t[63] = 1.99897152;
        t[64] = 1.99834054;
        t[65] = 1.99772965;
        t[66] = 1.99713791;
        t[67] = 1.99656442;
        t[68] = 1.99600835;
        t[69] = 1.99546893;
        t[70] = 1.99494542;
        t[71] = 1.99443711;
        t[72] = 1.99394337;
        t[73] = 1.99346357;
        t[74] = 1.99299713;
        t[75] = 1.99254350;
        t[76] = 1.99210215;
        t[77] = 1.99167261;
        t[78] = 1.99125440;
        t[79] = 1.99084707;
        t[80] = 1.99045021;
        t[81] = 1.99006342;
        t[82] = 1.98968632;
        t[83] = 1.98931856;
        t[84] = 1.98895978;
        t[85] = 1.98860967;
        t[86] = 1.98826791;
```

```
        t[87] = 1.98793421;
        t[88] = 1.98760828;
        t[89] = 1.98728986;
        t[90] = 1.98697870;
        t[91] = 1.98667454;
        t[92] = 1.98637715;
        t[93] = 1.98608632;
        t[94] = 1.98580181;
        t[95] = 1.98552344;
        t[96] = 1.98525100;
        t[97] = 1.98498431;
        t[98] = 1.98472319;
        t[99] = 1.98446745;
        t[100] = 1.98421695;
        t[101] = 1.98397152;
        t[102] = 1.98397152;
        t[103] = 1.98397152;
        t[104] = 1.98397152;
        t[105] = 1.98397152;
        t[106] = 1.98397152;
        t[107] = 1.98397152;
        t[108] = 1.98397152;
        t[109] = 1.98397152;
        t[110] = 1.98397152;
        t[111] = 1.98397152;
        t[112] = 1.98397152;
        t[113] = 1.98397152;
        t[114] = 1.98397152;
        t[115] = 1.98397152;
        t[116] = 1.98397152;
        t[117] = 1.98397152;
        t[118] = 1.98397152;
        t[119] = 1.98397152;
        t[120] = 1.98397152;
        t[121] = 1.98397152;
        t[122] = 1.98397152;
        t[123] = 1.98397152;
        t[124] = 1.98397152;
        t[125] = 1.98397152;
        t[126] = 1.98397152;
        t[127] = 1.98397152;
        t[128] = 1.98397152;
        t[129] = 1.98397152;
        t[130] = 1.98397152;
        t[131] = 1.98397152;
        t[132] = 1.98397152;
        t[133] = 1.98397152;
        t[134] = 1.98397152;
        t[135] = 1.98397152;
        t[136] = 1.98397152;
        t[137] = 1.98397152;
        t[138] = 1.98397152;
        t[139] = 1.98397152;
        t[140] = 1.98397152;
        t[141] = 1.98397152;
        t[142] = 1.98397152;
        t[143] = 1.98397152;
        t[144] = 1.98397152;
        t[145] = 1.98397152;
        t[146] = 1.98397152;
        t[147] = 1.98397152;
```

```
        t[148] = 1.98397152;
        t[149] = 1.98397152;
        t[150] = 1.98397152;
        t[151] = 1.98397152;
        t[152] = 1.98397152;
        t[153] = 1.98397152;
        t[154] = 1.98397152;
        t[155] = 1.98397152;
        t[156] = 1.98397152;
        t[157] = 1.98397152;
        t[158] = 1.98397152;
        t[159] = 1.98397152;
        t[160] = 1.98397152;
        t[161] = 1.98397152;
        t[162] = 1.98397152;
        t[163] = 1.98397152;
        t[164] = 1.98397152;
        t[165] = 1.98397152;
        t[166] = 1.98397152;
        t[167] = 1.98397152;
        t[168] = 1.98397152;
        t[169] = 1.98397152;
        t[170] = 1.98397152;
        t[171] = 1.98397152;
        t[172] = 1.98397152;
        t[173] = 1.98397152;
        t[174] = 1.98397152;
        t[175] = 1.98397152;
        t[176] = 1.98397152;
        t[177] = 1.98397152;
        t[178] = 1.98397152;
        t[179] = 1.98397152;
        t[180] = 1.98397152;
        t[181] = 1.98397152;
        t[182] = 1.98397152;
        t[183] = 1.98397152;
        t[184] = 1.98397152;
        t[185] = 1.98397152;
        t[186] = 1.98397152;
        t[187] = 1.98397152;
        t[188] = 1.98397152;
        t[189] = 1.98397152;
        t[190] = 1.98397152;
        t[191] = 1.98397152;
        t[192] = 1.98397152;
        t[193] = 1.98397152;
        t[194] = 1.98397152;
        t[195] = 1.98397152;
        t[196] = 1.98397152;
        t[197] = 1.98397152;
        t[198] = 1.98397152;
        t[199] = 1.98397152;
        t[200] = 1.98397152;
   }
   if (ic==99) {
       t[20] = 2.861;
       t[15] = 2.977;
       t[10] = 3.25;
       t[5] = 4.604;
       t[3] = 9.925;
   }
```

```
   sort = "sort -n";
   for (x in xvalues) {
       for (j=1;j<=nrvar;j++) {
           for (i=1;i<=amostras[x];i++) {
               mv[x,j] = mv[x,j] + (v[x,i,j]/amostras[x]);
           }
           for(i=1;i<=amostras[x];i++){
               var_v[x,j] = var_v[x,j] + ((v[x,i,j] -
mv[x,j])^2)/(amostras[x]-1);
           }
           inter_v[x,j] = t[amostras[x]] * sqrt(var_v[x,j]/amostras[x]);
       }

       printf("%3.6f\t",x) | sort;

       for (j=1;j<=nrvar;j++) {

           printf("%2.8f\t%2.8f\t", mv[x,j], inter_v[x,j]) | sort;
       }
       printf("\n") | sort ;
   }
   close(sort);
}
```

7.3.3 analise_ping_perda.awk

```
#!/usr/bin/awk -f

BEGIN {
        tot = 0 ;
}


{
   if ($7 == "packet") {
        print size " " $6
}

   if ($9 == "packet") {
        print size " " $8
}

}
END {
}
```

7.3.4 analise_ping_troca.awk

```
#!/usr/bin/awk -f

BEGIN {
        count = 0 ;
```

```
}


{
  if ($1 == "RR:") {
        count = count + 1
}


}
END {
print size " " count
}
```