MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMUNICAÇÕES

1º Ten PEDRO LOAMI BARBOSA
1º Ten LUCAS TADEU STUDART DE CARVALHO
1º Ten DOUGLAS MARREIRA DOS SANTOS

DESENVOLVIMENTO DE PROTOCOLO DE ROTEAMENTO PARA REDES WIRELESS AD-HOC

Rio de Janeiro
2015

INSTITUTO MILITAR DE ENGENHARIA

1º Ten PEDRO LOAMI BARBOSA
1º Ten LUCAS TADEU STUDART DE CARVALHO
1º Ten DOUGLAS MARREIRA DOS SANTOS

DESENVOLVIMENTO DE PROTOCOLO DE ROTEAMENTO PARA REDES WIRELESS AD-HOC

Projeto de Final de Curso apresentado ao Curso de Graduação em Engenharia de Comunicações do Instituto Militar de Engenharia.

Orientador: Vítor Gouvêa Andrezo Carneiro – D.C.
Co-orientador: Sérgio dos Santos Cardoso Silva – M.C.

Rio de Janeiro
2015

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 – Praia Vermelha
Rio de Janeiro – RJ CEP: 22290-270

Protocolos de Redes Ad Hoc. Pedro Loami Barbosa; Lucas Tadeu Sturdat; Douglas Marreira dos Santos. - Rio de Janeiro: Instituto Militar de Engenharia, 2015.
Projeto de Final de Curso – Instituto Militar de Engenharia, 2015.

INSTITUTO MILITAR DE ENGENHARIA

1º Ten PEDRO LOAMI BARBOSA
1º Ten LUCAS TADEU STUDART DE CARVALHO
1º Ten DOUGLAS MARREIRA DOS SANTOS

DESENVOLVIMENTO DE PROTOCOLO DE ROTEAMENTO PARA REDES WIRELESS AD-HOC

Projeto de Final de Curso apresentado ao Curso de Graduação em Engenharia de Comunicações do Instituto Militar de Engenharia.

Orientador: Vítor Gouvêa Andrezo Carneiro – DsC
Co-orientador: Sérgio dos Santos Cardoso Silva – Ms

Aprovada em 28 de junho de 2015 pela seguinte Banca Examinadora:

_____
Prof. Vítor Gouvêa Andrezo Carneiro – D.C. do IME

_____
Prof. Sérgio dos Santos Cardoso Silva – M.C. do IME

_____
Prof. André Luis Souza de Araújo – M.C. do IME

Rio de Janeiro
2015

Sumário

RESUMO

A tecnologia das redes sem fio são tecnologias bastante presentes em qualquer ambiente atualmente. Aparelhos portáteis são cada vez mais comuns na vida das pessoas e, para manter sua conectividade, seja com a Internet ou entre os próprios dispositivos pessoais, tecnologias de redes sem fio tem sido criadas e melhoradas. Algumas destas tecnologias se baseiam no tipo de topologia de redes sem fio Ad Hoc, como por exemplo o bluetooth, onde os dispositivos se conectam uns aos outros independentemente de uma estrutura fixa ou nó central. As redes Ad Hoc possuem uma grande variedade de aplicações, desde aplicativos para celulares e automação residencial, até aplicações militares, como o acompanhamento de tropas em campo.

Este projeto tem por finalidade desenvolver e analisar um protocolo de roteamento para redes sem fio Ad Hoc com múltiplos nós intermediários e propriedades de comutação e roteamento. Foi feito um estudo sobre o modo de operação de uma rede ad hoc, bem como de suas peculiaridades e foi realizado um estudo geral de protocolos utilizados para esses tipos de rede sem infraestrutura definida.

A partir das características de cada protocolo estudado, vantagens e desvantagens, optou-se pela implementação do *Dynamic Source Routing* (DSR), protocolo reativo que envolve conceitos como *Route Discovery* e *Route Maintenance* entre os dispositivos. Também foram estudadas as condições em que se processam mídias diversas que possam vir a trafegar pela rede, principalmente no que se refere a condições mínimas de operação para o tráfego efetivo e satisfatório de dados na rede. E por fim um experimento prático para mostrar os resultados da implementação, enviando mensagens a partir de um computador, verificando o roteamento feito pelo DSR e obervando a criação e manutenção das rotas.

Palavras-chave: Ad Hoc, *Dynamic Source Routing*, Redes móveis, *Route Discovery*, *Route Maintenance*, QoS.

1. INTRODUÇÃO

Atualmente, a Internet evidencia ser um meio cada vez mais intrínseco à sociedade. O desenvolvimento da tecnologia tem contribuído para o surgimento de uma era marcada por *smartphones*, computadores pessoais, tablets, e outros dispositivos. Os smartphones finalizam a época dos simples telefones celulares, agregando a estes inúmeras funcionalidades como câmeras fotográficas de alta resolução, jogos em alta definição, aplicativos para edição de documentos e imagens, entre outros. Computadores pessoais apresentam maior poder de processamento e, com isso, aumenta-se o rigor por experiências de usuário cada vez melhores dos dispositivos e das aplicações. Entretanto, a mudança maior está na capacidade de se integrarem de forma eficiente à Internet em tempo real. Possibilita-se, então, a troca instantânea de mensagens e aplicações interativas, em que milhares de usuários compartilham informações em tempo real; fornecem suas localizações geográficas; acessam seus e-mails; assistem a vídeos; participam de videoconferências; pagam suas contas bancárias; resumindo grande parte das atividades cotidianas por meio do acesso à rede e depositando suas confianças na eficiência de todo o processo transparente, formado pelos complexos protocolos que constroem a Internet e a interligação dos dispositivos. Além dissotodo esse processo do uso de cabos, graças as redes sem fio e posterior surgimento dos *hotspots*, os pontos de acesso à internet em locais públicos, diminuindo a lacuna entre os usuários e a rede.

O notável crescimento das redes sem fio tem levado a uma grande proliferação de tecnologias *wireless*. Diversos protocolos e padrões surgem com o objetivo de melhorar a qualidade da comunicação e ajudar na portabilidade de dispositivos sem fio mantendo a conectividade. Por exemplo, muitos dispositivos dentro de uma rede pessoal podem ser conectados via Bluetooth, tecnologia sem fio bastante utilizada para interligar celulares, computadores, fones de ouvido sem fio entre muitos outros equipamentos.

As redes de computadores em geral são projetadas para manter a conexão dos usuários durante todo o tempo, o que não é diferente para as redes sem fio, que além de garantir a conexão dos usuários deve permitir mobilidade e uma grande área de cobertura, aspectos que devem ser considerados no desenvolvimento de qualquer protocolo para rede sem fio (FOROUZAN, 2007). Outros atributos, como eficiência da entrega de dados, descobrimento e manutenção de rotas, tempo de convergência e recuperação de erros também são muito importantes na criação de um padrão para qualquer tipo de rede.

Os protocolos de redes ad hoc são desenvolvidos levando em consideração todos os aspectos para uma rede sem fio, mas também devem funcionar sem nenhuma estrutura fixa e todos os nós devem possuir funções de roteamento e/ou comutação, para que a rede possua um maior alcance. Assim a tarefa de um protocolo ad hoc pode

se tornar bastante complexa à medida que cresce o número de usuários e dispositivos conectados.

1.1. Motivação

Protocolos de redes são desenvolvidos e melhorados a todo tempo, devido à crescente demanda de conectividade e o surgimento de novas aplicações. Essas melhorias devem ser capazes de manter a Qualidade de Serviço (QoS), garantindo aplicações em tempo real, o tráfego de dados estável e uma banda considerável para uma grande quantidade de usuários e/ou dispositivos, essas melhorias devem ser capazes de suprir a necessidade de velocidade e banda entregues (TANENBAUM, 2002). Na motivação da criação de protocolos para redes Ad Hoc, além de todos os fatores relacionados ao QoS, existem diversas outras aplicações, como conexão de dispositivos domésticos, aplicações militares em aeronaves, viaturas e pessoal em campo.

1.2. Objetivo

Este projeto tem como objetivo desenvolver e analisar um protocolo de roteamento para redes sem fio ad hoc com múltiplos nós intermediários e propriedades de comutação e roteamento. Com base em um estudo de protocolos de roteamento de redes já existentes, escolheu-se um protocolo base, a partir de suas vantagens e desvantagens, desenvolver o protocolo e simular uma rede com suas características, analisando os resultados da comunicação entre os dispositivos.

1.3. Justificativa

O desenvolvimento de protocolos e o melhoramento dos já existentes são uma demanda constante em redes de computadores. O surgimento de novas aplicações, a necessidade de maiores velocidades de transmissão e de largura de banda levam ao aumento dessa demanda e consequentemente a criação de novas tecnologias.

A manutenção de conectividade em redes sem fio Ad Hoc, juntamente com um melhoramento do QoS levam ao desenvolvimento de protocolos de rede mais eficientes e robustos. Logo, diversos problemas de redes sem fio podem ser solucionados com um protocolo mais eficaz e melhorado.

1.4. Metodologia

Inicialmente será realizado um estudo referente aos conceitos relacionados a redes sem fio, dando enfoque especial a redes ad hoc, com o objetivo de, além de

conhecer o que já existe em termos de protocolos para esses tipos de rede, conhecer os parâmetros nos quais a implementação do protocolo escolhido irá se basear.

Também serão estudados detalhadamente os conceitos de Qualidade de Serviço (QoS) para redes sem fio, com o objetivo de especificar parâmetros necessários para esse tipo de rede.

Uma vez determinado o protocolo e os parâmetros a serem seguidos, será implementada em C++ uma versão básica do mesmo, que será analisada, com o objetivo de verificar se, de fato, ele atende às especificações de uma rede ad hoc.

1.5. Estrutura

No capítulo 2 deste trabalho são introduzidos conceitos básicos de redes ad hoc, tomando como base o padrão IEEE 802.11 e seus tipos de topologia. Neste capítulo, também são discutidos alguns problemas que os protocolos de redes sem fio encontram na busca de uma melhor eficiência.

No capítulo 3 são discutidos os conceitos relativos a Qualidade de Serviço, descrevendo os principais parâmetros e valores típicos dos mesmo para determinadas aplicações.

No capítulo 4 são apresentados os tipos de protocolos de redes ad hoc, descrevendo as principais categorias e as técnicas de roteamento. Concluindo com a escolha do protocolo que será analisado.

No capítulo 5, o protocolo DSR é analisado a fundo, apresentando todos os seus atributos, descrevendo as mensagens trocadas para obtenção e manutenção de rotas e detalhando as técnicas utilizadas pelo protocolo.

No capítulo 6 é desenvolvida a lógica de implementação do protocolo na linguagem C++, apresentando classes e algoritmos utilizados na programação.

No capítulo 7 é descrita a simulação feita como experimento prático a fim de comprovar a utilização do protocolo DSR em uma rede ad hoc.

O capítulo 8 contém a conclusão do trabalho quanto ao desenvolvido até o momento.

E o capítulo 9 aborda os possíveis trabalhos futuros a serem realizados a partir deste estudo.

2. INTRODUÇÃO A REDES AD HOC

O conceito de redes ad hoc surgiu na década de 1970, quando a U.S DARPA (*United States Defense Advanced Research Projects Agency*) iniciou o projeto *Packet Radio Network*, para explorar o uso de redes na comunicação de pacotes via rádio num ambiente tático militar. Mais tarde essas redes passaram a ser adaptáveis as rápidas mudanças que ocorrem em um ambiente tático.

Em telecomunicações, redes ad hoc é um tipo de topologia de rede que não possui um nó terminal, ou ponto de acesso, para onde as comunicações convergem e que as encaminha para os respectivos destinos. Assim, uma rede de computadores ad hoc é aquela na qual todos os terminais funcionam como roteadores, encaminhando de forma comunitária as comunicações provenientes dos terminais vizinhos. Como exemplo de um protocolo de redes ad hoc, temos o DSR (*Dynamic Source Routing*) que será discutido detalhadamente neste trabalho.

2.1. O protocolo IEEE 802.11

A norma IEEE 802.11 é um padrão internacional que descreve as características de uma rede local sem fio (WLAN). Atualmente, bastante utilizado em ambientes domésticos e escritórios, com garantia de interoperabilidade de equipamentos para a comunicação sem fio. Em sua última versão, 802.11ac, o protocolo é capaz de aproveitar de forma bastante eficiente a banda disponível, com um grande alcance e taxas que chegam até 1,3 Gbps.

O protocolo descreve o funcionamento das camadas mais baixas do modelo OSI para uma conexão sem fios que utiliza ondas eletromagnéticas. Para a camanda física, estabelece os tipos de codificação da informação e o modo de transmissão. Para a camada de enlace e suas subcamadas MAC e LLC, estabelece as regras de acesso ao meio de forma semelhante ao padrão Ethernet.

Existem dois modos de operação de uma rede baseada no padrão 802.11, o modo infraestruturado e o modo ad hoc. As figuras abaixo mostram o funcionamento básico de cada tipo de topologia.

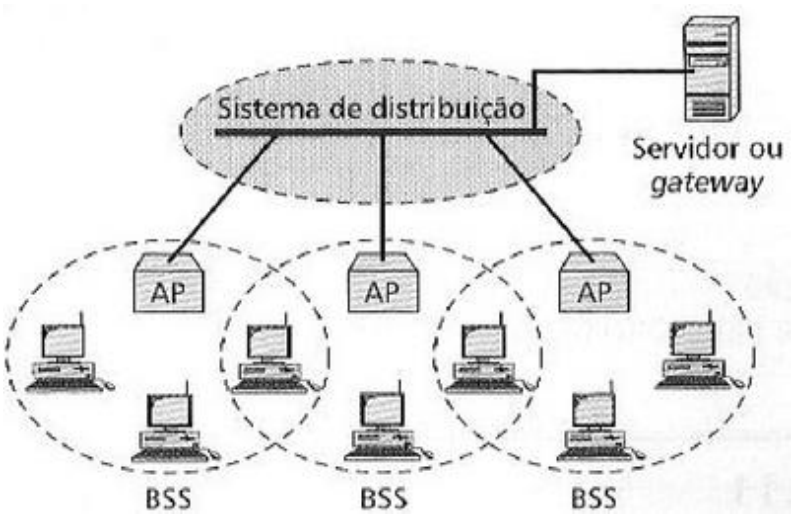

Figura 1 - 802.11 - Modo Infraestruturado

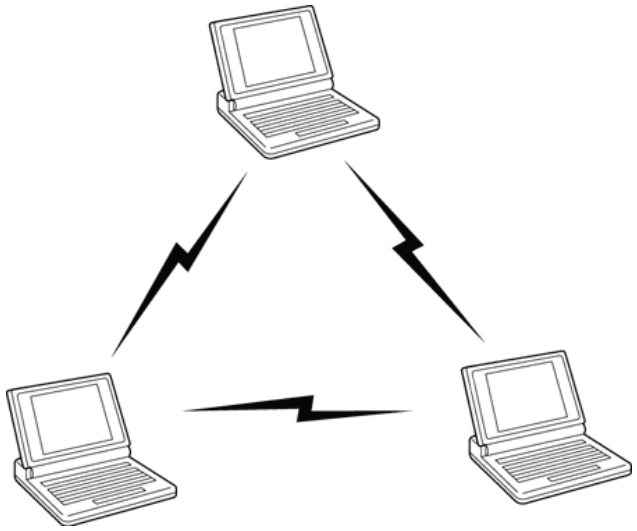
Figura 2 - 801.11 - Modo Ad Hoc

No modo infraestruturado, existe uma rede fixa que distribui os pontos de acesso da rede por toda a área de serviço. Essa rede fixa, ou sistema de distribuição, é responsável por conectar todos os *Basic Service Set*, ou BSS, região de alcance de um ponto de acesso, entre eles mesmos e com a rede externa. O conjunto formado pelas áreas de serviço de cada AP (*Access Point*) é chamado de *Extended Service Set*, ou ESS. Assim, esse tipo de topologia depende de uma estrutura fixa para que a comunicação entre os nós aconteça. Já o modo ad hoc, como comentado anteriormente, independe de uma estrutura fixa ou ponto de acesso para que ocorra a comunicação.

Para redes ad hoc, o 802.11 descreve os detalhes do que acontece nas camadas mais baixas durante a comunicação, assim protocolos para esse tipo rede

apenas se preocupam com a comunicação lógica, enquanto o 802.11 garante a comunicação física e o acesso ao meio.

2.2. Problemas das redes sem fio

As redes sem fio possuem problemas peculiares devido à sua forma de transmissão dos dados. O espectro eletromagnético é bastante disputado por muitos equipamentos sem fio, para estas redes o controle acesso ao meio é realizado pelo protocolo 802.11 utilizando a técnica do CSMA/CA. Esta técnica consiste em evitar as colisões[1] na disputa de acesso ao meio. Um dos principais problemas para o qual essa técnica é utilizada é o do terminal escondido. Quando dois nós tentam transmitir para um terceiro nó, sendo que os dois primeiros não estão no alcance um do outro, pode ocorrer colisão sem que os nós transmissores percebam. O CSMA/CA trata deste problema, sendo utilizado para qualquer tipo de topologia.

Por terem características bastante particulares, as redes adhoc devem considerar com maior rigor as observações a respeito das métricas de QoS, em se tratando de acompanharem o atual padrão de transferência de dados das comunicações sem fio. Por não apresentarem uma infraestrutura, essas redes estão sujeitas principalmente a dificuldades de controle e segurança. Assim não há elemento central que realize o monitoramento da rede e de todas as suas entregas de dados entre os dispositivos. Essa ausência requer naturalmente uma relação de compromisso entre a flexibilidade na gerência da rede e a complexidade da implementação dos protocolos referentes a ela, para compensar a não centralidade das informações, sendo resolvida por meio de colaboração entre os nós na escolha de rotas e na alocação de recursos.

Os ambientes dinâmicos dos dispositivos, que são traduzidos nos movimentos dos usuários nos locais de acesso à rede, constantemente variam a sua topologia e dificultam ainda mais qualquer implementação que vise o aperfeiçoamento dessas redes. Daí, surgem técnicas de reroteamento de pacotes, uma vez que nós anteriormente presentes agora podem ceder lugar a outros nós, ou mesmo deixarem de integrar um caminho válido para a entrega de um conjunto de pacotes. Isso gera variações de carga na rede e a consequente realocação de seus recursos, podendo ocasionar congestionamentos transientes, e causar impactos severos na qualidade de serviços oferecidos por esse sistema. Além disso, fatores preponderantes das redes ad hoc, como limitações dos quesitos de energia (por serem dispositivos móveis e fazerem uso de baterias), largura de banda disponível, poder de processamento e densidade da rede são de suma importância quando da implementação dessas redes não infraestruturadas.

[1] Colisão: termo utilizado quando dois ou mais nós tentam transmitir ao mesmo tempo

Muito comum em redes ad hoc, e como resultado da volatilidade da topologia, a ocorrência de nós congestionados pode resultar em perda de pacotes, aumento de *delay* fim a fim e, por vezes, necessitam de técnicas para o estabelecimento de rotas de manutenção, visando evitar ou mesmo minimizar problemas desse tipo, objetivando a operacionalidade da rede e, em plano pouco menos relevante, impedir a degradação dos serviços.

3. PROTOCOLOS DE ROTEAMENTO EM REDES AD HOC

O roteamento em redes de computadores é dado pelo processo de envio, encaminhamento e entrega de pacotes de dados dentro da rede. Muitas vezes os protocolos de roteamento se baseiam em tabelas usadas para mapear a topologia, estabelecer e desfazer rotas e encaminhar os pacotes de forma correta até seu destino. A construção, atualização e manutenção variam de acordo com o método de roteamento escolhido[2].

Devido a dinamicidade de uma rede ad hoc, os protocolos desenvolvidos para redes cabeadas comuns se mostram ineficientes nesse tipo de rede. Fatores como o consumo de energia e banda limitam a eficiência do roteamento, tornando mais complexo o desenvolvimento de algoritmos mais efetivos.

A implementação de muitos protocolos para redes ad hoc levou à divisão destes em categorias baseadas nos algoritmos utilizados em descobrimento, manutenção e atualização de rotas. Na separação das categorias são levados em conta fatores relevantes como bateria, utilização de banda e localização. São elas: proativos, reativos, híbridos, *location-aware* e *energy-aware*. Sendo os dois últimos não detalhados neste trabalho.

3.1. Protocolos proativos

Esta categoria reúne os protocolos que mantêm e atualizam tabelas de roteamento constantemente. Informações de rotas, endereços destinos e gateways são armazenados em memória para consulta no momento da transmissão de um pacote. Para a manutenção de rotas, mensagens de atualização são enviadas regularmente com informações de novas rotas e rotas que deixaram de funcionar. Como principais exemplos de protocolos proativos temos o *Destination-Sequenced Distance-Vector* (DSDV) e o *Wireless Routing Protocol* (WRP).

O maior problema enfrentado por protocolos proativos é a quantidade de mensagens na rede. Todas as mensagens de atualização de rotas que são enviadas constantemente congestionam a rede. As tabelas de roteamento devem ser atualizadas a todo momento com informações como novos vizinho de um nó, rotas duplicadas que geram *loops*, rotas que deixaram de funcionar porque o destino saiu do alcance de qualquer elemento da rede, entre outros tipos de atualização. Tudo isso gera mensagens que são enviadas, na maioria das vezes, em *broadcast*, congestionando ainda mais a rede.

Podemos concluir que protocolos proativos são bastante eficientes na manutenção de rotas para redes com número de nós reduzido, pois assim a quantidade de mensagens de atualização vai ser limitada a poucos nós e, com a

[2] Texto pode ser encontrado em: http://www.gta.ufrj.br/grad/09_1/versao-final/adhoc/protocolos.html

manutenção de rotas, o atraso na entrega de dados vai diminuir, pois cada nó sempre saberá o melhor caminho para chegar ao destino.

3.2. Protocolos reativos

O conjunto de protocolos reativos são caracterizados pela atuação sob demanda quando se trata da transmissão. O grande fluxo de mensagens de atualização de rotas é inexistente. As rotas são descobertas a cada tentativa de transmissão, com uma mensagem de pedido de rota (*Route Request*). Assim as tabelas de roteamento apenas são atualizadas quando houver qualquer tipo de transmissão de dados. Daí a característica reativa dessa categoria, a rede só reage a uma mudança na topologia quando há uma transmissão mal sucedida, sendo necessária uma nova descoberta de rota. Como principais exemplos de protocolos reativos temos o *Ad Hoc On-Demand Distance Vector* (AODV) e o *Dynamic Source Routing* (DSR).

Como principais desvantagens dos protocolos reativos temos o atraso de entrega de dados devido ao tempo necessário para a descoberta de uma rota ou uma recuperação de rota quebrada. Como as tabelas de rota não estão sempre atualizadas, no momento da transmissão podem existir rotas guardadas em cache que deixaram de funcionar devido à mobilidade da rede. Por outro lado, há pouco congestionamento da rede com mensagens de atualização.

Podemos concluir que os protocolos reativos são bastante eficientes na transmissão de dados em redes com baixa mobilidade, pois assim as rotas guardadas em cache não irão deixar de funcionar tão depressa. Redes bastante reativas tendem a gerar menos tráfego reduzindo também a perda de pacotes devido a interferências.

3.3. Protocolos híbridos

Esta categoria pode ser descrita como um conjunto de protocolos que utilizam de características de ambas as categorias anteriores, proativos e reativos. Os protocolos híbridos aproveitam as vantagens de proatividade em algumas situações com características reativas para outras. O principal exemplo de protocolo híbrido é o *Zone Routing Protocol*. Outros protocolos híbridos se baseiam nas técnicas utilizadas no ZRP no desenvolvimento.

A separação entre características proativas e reativas é geralmente visualizada ao se definir uma área de atuação de cada tipo. Por exemplo, no ZRP esta área é definida como um raio a partir de um determinado nó, ou raio da zona, outro nó que estiver a uma distância, em saltos (*hops*), menor ou igual ao raio da zona estará dentro da área de atuação da parte proativa relativa aquele nó. Ou seja, quem estiver dentro da zona de um determinado nó trocará mensagens de atualização de rotas com o nó central. E quem estiver fora da zona de um determinado nó funcionará de forma reativa em relação ao referido nó, ou seja, qualquer comunicação para fora da zona vai

acontecer de forma reativa. Dessa forma o ZRP aproveita as vantagens da proatividade dentro da zona de cada nó e trabalha reativamente, caso o destino da comunicação esteja fora da zona.

Assim a principal dificuldade encontrada em uma rede híbrida é a implementação desse tipo de protocolo. A complexidade do algoritmo torna sua implementação desnecessária para certas aplicações.

Após um estudo detalhado sobre os tipos de protocolos e do funcionamento de cada categoria, o protocolo DSR foi escolhido para ser implementado como objetivo deste trabalho. A opção se baseou pelo motivo de uma menor complexidade de desenvolvimento e a documentação (RFC 4728) detalha o funcionamento do DSR de forma bem consistente, ao contrário de outros protocolos que por falta de fontes de consulta mais aprofundadas não poderiam ser implementados facilmente.

4. O PROTOCOLO DSR

O protocolo DSR, ou *Dynamic Source Routing*, é um protocolo simples e eficiente para uso específico em redes wireless *multi-hop* e redes ad hoc, sendo capaz de responder prontamente a variações na topologia da rede. O protocolo permite a autoconfiguração e manutenção da rede e não requer que seja infraestruturada, isto é, moderada por um Sistema de Distribiução. Assim alterações na rede são automaticamente detectadas e corrigidas pelo protocolo, que é composto basicamente de dois mecanismos principais para estabelecimento e manutenção de caminhos entre os nós (*hosts*), o *Route Discovery* e o *Route Maintenance* respectivamente. O protocolo, apesar de propor eficiência na comunicação de nós móveis, apresenta melhor desempenho para baixa velocidade e mobilidade dos nós, embora ainda apresente desempenho satisfatório para um número considerável deles (aproximadamente 200).

Cada elemento da rede armazena um cache de roteamento com rotas conhecidas por si, e que são atualizadas por meio dos processos citados anteriormente de descoberta e manutenção de rotas, valendo lembrar que é um protocolo reativo, isto é, espera-se que uma rota seja requisitada para que sejam buscadas informações da rede, por meio da cooperação entre os nós, e ela seja estabelecida. Logo, não há troca periódica de informações sobre a topologia da rede, sendo, portanto, um processo por demanda.

Por ser um protocolo reativo a alta mobilidade dos nós implica em maiores chances de os caminhos se tornarem inválidos e, portanto, maiores são as necessidades de estabelecimento de novas rotas, ou rotas de manutenção, diminuindo a eficiência do processo. Por outro lado, o fato de o protocolo permitir múltiplas rotas para qualquer destino auxilia na solução da dinamicidade do sistema, o que auxilia no balanço de cargas e no aumento da robustez do sistema, aliado à capacidade de cada nó poder selecionar e controlar seus caminhos para roteamento de seus pacotes.

4.1. Route Discovery

O *Route Discovery* é um mecanismo de descobrimento de rotas pelo qual um nó fonte (S) descobrirá uma rota para um nó destino (D) e é utilizado apenas quando aquele ainda não conhece uma rota para este. Um outro conceito necessário para entendimento desse mecanismo é o de *Route Cache*, ou seja, o armazenamento de uma lista de caminhos aprendidos por cada nó atráves de pacotes que trafegam por ele. Assim, quando um nó fonte (S) deseja enviar um pacote para o nó destino (D), o nó S primeiramente procura em sua *Route Cache* se possui uma rota válida para D. Se houver, ele a utiliza; caso contrário, o nó S utiliza o mecanismo de *Route Discovery* para encontrar uma rota. Nesse meio tempo, outros pacotes também são processados, sendo o descobrimento de rotas e o processamento de pacotes processos paralelos.

Por exemplo, supondo a necessidade de envio de um novo pacote de S para D, o nó S coloca um cabeçalho *Source Route*[3]. A Source Route é descoberta consultando-se a *Route Cache*, com rotas aprendidas anteriormente. Caso não haja essas informações em cache, o nó S inicia *Route Discovery* para encontrar novo caminho, iniciando pela solicitação – em *broadcast* – de um *Route Request.*

Adiante será reiteradamente utilizado o termo *option*, que deve ser entendido como o *payload* do cabeçalho DSR. Na Figura 3 encontra-se o padrão de *option* para o *Source Route*:

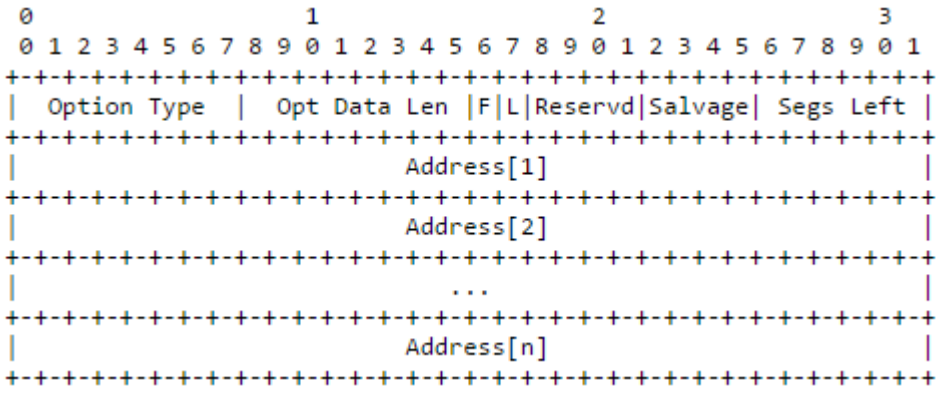

Figura 3 - Option do Source Route

Option Type: composto de 8 bits. Seu primeiro bit representa se um nó que recebe este pacote deverá responder com uma *Route Error* do tipo OPTION_NOT_SUPPORTED, exceto que uma *Route Error* jamais deverá ser enviada a um pacote contendo opção do tipo *Route Request*. Os dois bits seguintes descrevem como o pacote deve ser processado pelo nó que não suporta essa *Option Type*. Demais bits descrevem se se trata de *Route Request*, *Route Reply*, *Route Error*, *Acknowledgement Request*, *Acknowledgement*, *DSR Source*, e outros. Para o caso do *Option Type* do *Source Route*, pacotes que não compreenderem essa opção farão o descarte do pacote.

Opt Data Len: inteiro de 8 bits sem sinal. Comprimento do *option*, em octetos, excluindo os campos *Option Type* e *Opt Data Len*. Para o formato do option do DSR *Source Route* aqui definido, esse campo deve ser setado com o valor $(n * 4) + 2$, em que n é o número de endereços presentes nos campos Address[i].

First Hop External (F): usado para indicar que o primeiro salto indicado pelo *option* do DSR Source *Route* é, na verdade, um caminho arbitrário em uma rede externa à rede DSR. A rota exata realizada fora da rede DSR não é representada no *option* do DSR *Source Route*. Nós que armazenarem este salto em suas *Route Caches* deverão sinalizar o salto com a flag *External*. Esses saltos não devem ser retornados

[3] É o conjunto de saltos para chegar ao nó destino (D).

em um RRep gerado a partir das entradas desse *Route Caches*, e seleção de rotas a partir do *Route Cache* para rotear um pacote deverá preferir rotas que não contenham a flag *External* sinalizada.

Last Hop External (L): usado para indicar que o último salto indicado pelo *option* do DSR *Source Route* é, na verdade, um caminho arbitrário em uma rede externa à rede DSR. Informações adicionais são análogas ao tópico anterior.

Reserved: deve ser enviado como 0 e ignorado na recepção.

Salvage: inteiro de 4 bits sem sinal. Contador do número de vezes que um pacote foi salvo devido ao roteamento do protocolo DSR.

Segments Left (Segs Left): número de segmentos de rota que ainda faltam, isto é, número de nós intermediários listados explicitamente para ainda serem alcançados antes de se alcançar o destino final.

Address[1..n]: é a sequência de endereços da *Source Route*. O número de endereços presentes nos campos Address[1..n] é indicado pelo campo *Opt Data Len* com $n = (Opt\ Data\ Len - 2)/4$. No processo de encaminhamento de pacotes ao longo da DSR *Source Route*, utilizando o option do DSR *Source Route* no cabeçalho do *DSR Options*, o campo *Destination Address* no cabeçalho do pacote IP é sempre colocado como o último endereço a ser alcançado pelo pacote. Um nó que receba um pacote contendo um cabeçalho do *DSR Options* com um *option* de uma *DSR Source Route* deve examinar a *Source Route* indicada para determinar se esse é o próximo salto pretendido para o pacote e como será trafegado adiante.

4.1.1. Route Request

O *Route Discovery* inicia com a propagação de um *Route Request* (RReq), em *broadcast*, pelo nó iniciador para se descobrir uma rota para o destino (D). Feito isso, nós vizinhos ao iniciador recebem a solicitação. Cada RReq identifica o iniciador (S) e o alvo (D) da RReq e contém uma única identificação, determinada pelo iniciador. À medida que a *Route Request* é propagada pela rede, a ela é integrada uma lista dos nós intermediários pelos quais trafega.

Dito isso, algumas possibilidades são observadas para os nós que recebem a RReq:

- O nó que recebe a RReq é o próprio destino da RReq:

Nesse caso, ele retorna *Route Reply* para o nó iniciador, com uma cópia da lista dos nós intermediários pelos quais a RReq trafegou.

- O nó recebe uma RReq repetida, com mesmo iniciador (S) e alvo (D), ou seu endereço está no *route record*[4] da RReq recebida:

[4] É uma lista dos endereços de cada nó intermediário pelos quais a RReq trafega. É iniciado como uma lista vazia pelo nó iniciador.

Aqui, o nó descarta a solicitação recebida. Se for o caso de o endereço já estar no cache, o descarte da solicitação é claro exemplo de detecção de loop (solucionado pelo descarte da solicitação).

- O nó que recebe a RReq ainda não a recebeu e seu endereço não está no *route record* da RReq recebida:

Se houver rota para o alvo (D) em sua *Route Cache* ele retorna *Route Reply* (RRep) para o iniciador (S), concatenando a rota inicial com a rota encontrada no cache. Feito isso, verifica a existência de nós duplicados no caminho resultante confeccionado. Por outro lado, se não possuir rota para o alvo em sua *Route Cache*, adiciona o próprio endereço no *route record* do RReq e envia em novo *broadcast*, com mesma Id anterior.

A seguir, é mostrado o padrão de opção (*option*) de um *Route Request*:

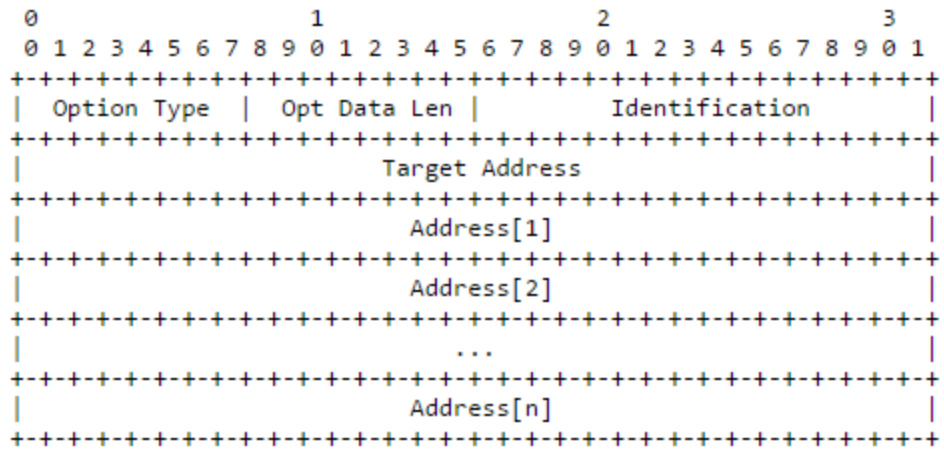

Figura 4 - Option do Route Request.

Antes de preenchido o option do *Route Request*, salienta-se que no preenchimento do cabeçalho IP devem constar nos campos *Source Address, Destination Address* e *Hop Limit* o seguinte:

Source Address: deve conter o endereço do nó que origina o *Route Request*. Nós intermediários que retransmitam o pacote em propagação do RReq não devem alterar esse campo.

Destination Address: deve ser o endereço de *broadcast* da rede

Hop Limit (TTL): pode assumir valores de 0 a 255, para implementar, por exemplo, *Route Requests* e *Route Requests* expandidos.

Uma explicação dos campos do option da *Route Request* são a seguir detalhados:

Option Type: idem à explicação fornecida para o *Source Route*.

Opt Data Len: inteiro sem sinal de 8 bits. É o comprimento do *option*, em octetos, excluindo os campos *Option Type* e *Opt Data Len*. Deve ser igual a (4 * n) + 6, em que n é o número de endereços no *option* do *Route Request*.

Identification: valor gerado pelo nó iniciador da *Route Request*. Esse valor é incrementado a cada RReq adicional com mesmos nós fonte e destino. Permite que um nó que receba o pacote seja capaz de saber se está recebendo uma solicitação repetida. Se esse valor é encontrado pelo nó em sua *Route Request Table*, então o nó deve descartar o pacote.

Target Address: é o endereço do nó alvo da *Route Request*.

Address[1..n]: são os endereços IPv4 de cada nó armazenado na *Route Request*. O endereço do *Source Address* no cabeçalho IP é o endereço do nó iniciador da *Route Discovery* e não deve ser colocado em nenhum dos campos de Address[1..n]. Address[1] é o endereço do primeiro nó após o nó iniciador. O número de endereços deste campo é indicado pelo campo Opt Data Len em que $n = (Opt\ Data\ len - 6)/4$. Cada nó que propague a RReq adiciona seu próprio endereço a essa lista, incrementando o *Opt Data Len* em quatro octetos.

O esquema da Figura 5 resume o processo do *Route Discovery* e do *Route Request*:

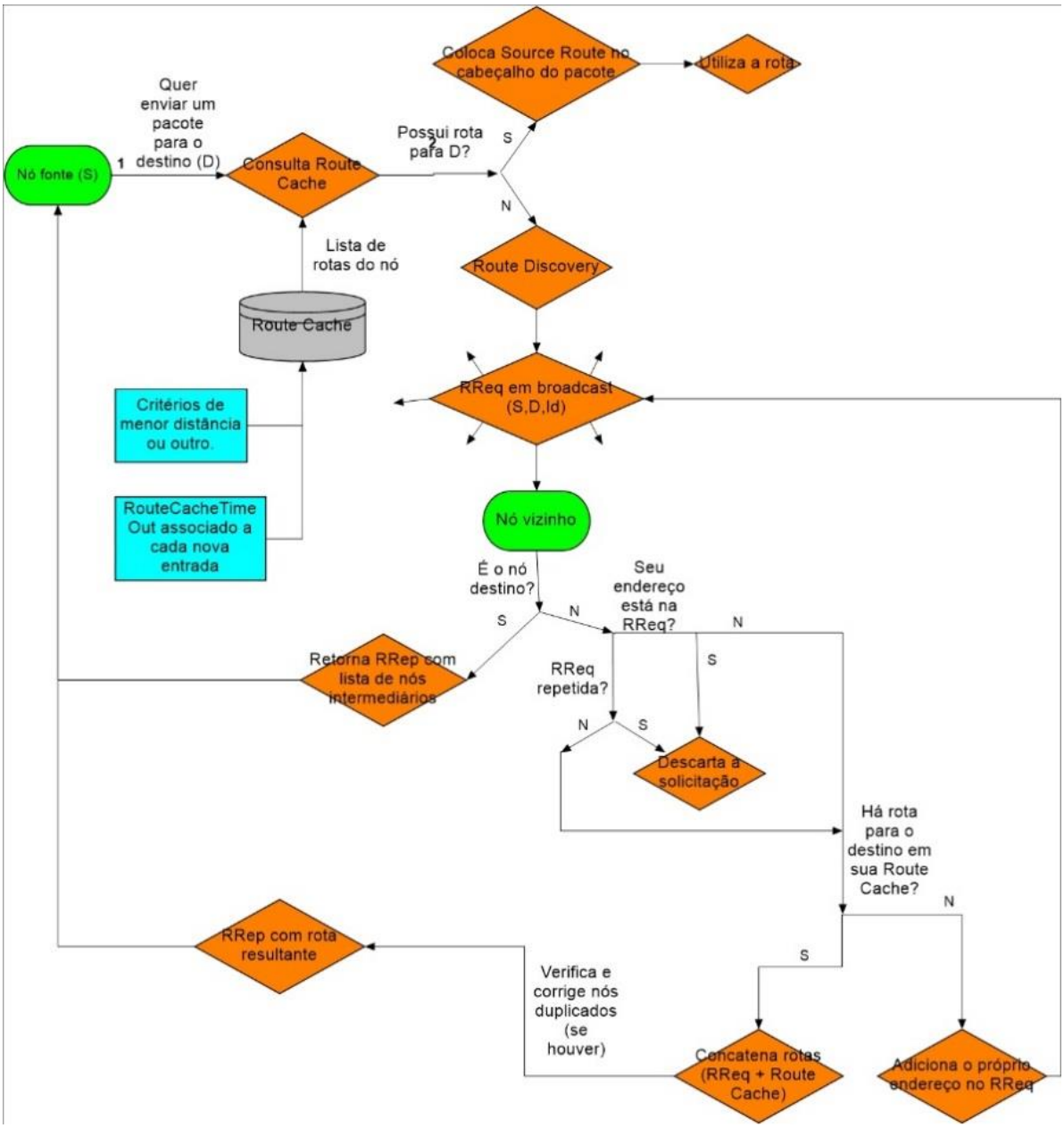

Figura 5 - Esquemático do Route Discovery

Suponha que um nó A deseje transmitir para um nó C e que o nó B seja um nó intermediário. Suponha ainda que nem A e nem B possuam rotas para C. De acordo com o protocolo, A iniciará o processo com um *Route Request* em *broadcast.* B receberá essa mensagem, e como também não possui rotas para C ele irá colocar seu endereço como um intermediário e retransmitirá. Note que ao retransmitir de B, A receberá a mesma *Route Request* que originou. Para evitar que esse problema ocorra indefinidamente, existe uma tabela chamada *Route Request Table* que armazena todos os *Route Requests* enviados por um determinado nó por um certo período de tempo.

Assim, quando um nó recebe uma mensagem desse tipo ele a descarta pois verifica que ela ainda se encontra na tabela de *Route Request*.

4.1.2. Route Reply

A *Route Reply* (RRep) é uma mensagem de resposta ao nó iniciador (S) quando se encontra o alvo (ou destino,D), e pode ser fornecida pelo próprio D, ou então por nó intermediário que conheça rota para ele. Dessa forma, algumas possibilidades são encontradas para o envio da RRep:

- O nó examina sua *Route Cache*:

Se encontrar rota de volta para o iniciador, e for permitida comunicação bidirecional no sistema, ele utiliza a o caminho reverso do RReq para entregar o RRep. Caso contrário, elabora sua própria *Route Discovery* em busca do iniciador (S), enviando sua RReq com o RRep, este em *piggybacking*[5].

Já ao recebimento da RRep pelo nó iniciador, procede-se com o armazenamento em cache da rota do RReq correspondente para posterior envio de pacotes para aquele destino, se for necessário.

A seguir é fornecido o padrão do *option* do *Route Reply*:

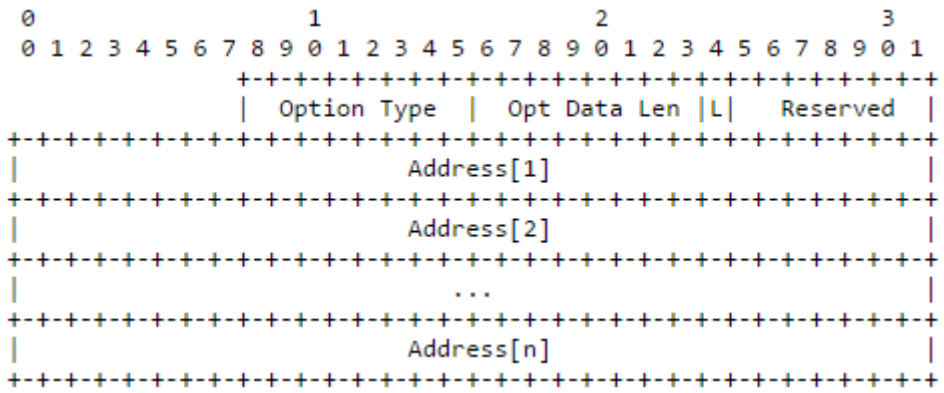

Figura 6 – Padrão do *option* do Route Reply

Antes de preenchido o *option* do RRep, é necessário o preenchimento do cabeçalho IP:

Source Address: deve ser colocado o endereço do nó que envia o *Route Reply*.

Destination Address: deve ser o endereço do nó fonte da rota a ser retornada. Copiado do campo de *Source Address* do *Route Request* gerador do *Route Reply*

Uma explicação dos campos do *option* do *Route Reply* são a seguir detalhados:

Option Type: idem à explicação fornecida para o *Route Request*.

[5] Técnica utilizada para incrementar a eficiência de protocolos bidirecionais. Cita-se, como exemplo, o envio de um quadro contendo informações de A para B, mas que carrega também informações de controle sobre quadros anteriores bem sucedidos (ou perdidos) de B

Opt Data Len: inteiro de 8 bits sem sinal. Comprimento do *option*, em octetos, excluindo os campos *Option Type* e *Opt Data Len*. Deve ser igual a $(4 * n) + 1$, em que n é o número de endereços no option do *Route Reply*.

Last Hop External (L): serve para indicar que o último salto dado pelo *Route Reply* é, na verdade, um caminho arbitrário em uma rede externa à rede DSR. A rota exata fora da rede DSR não é representada no *Route Reply*.

Reserved: deve ser enviado como 0 e ignorado na recepção.

Address[1..n]: é a *Source Route* retornada no *Route Reply*. Indica uma sequência de saltos, originado no nó fonte especificado no campo *Destination Address* do cabeçalho IP do pacote que trafega o *Route Reply*, através dos Address[i] nós na ordem listada no RRep, terminando no nó indicado por Address[n]. O número de endereços nos campos Address[1..n] é indicado pelo campo *Opt Data Len* com $n = (Opt\ Data\ len - 1)/4$.

A Figura 7 resume o processo do *Route Reply*:

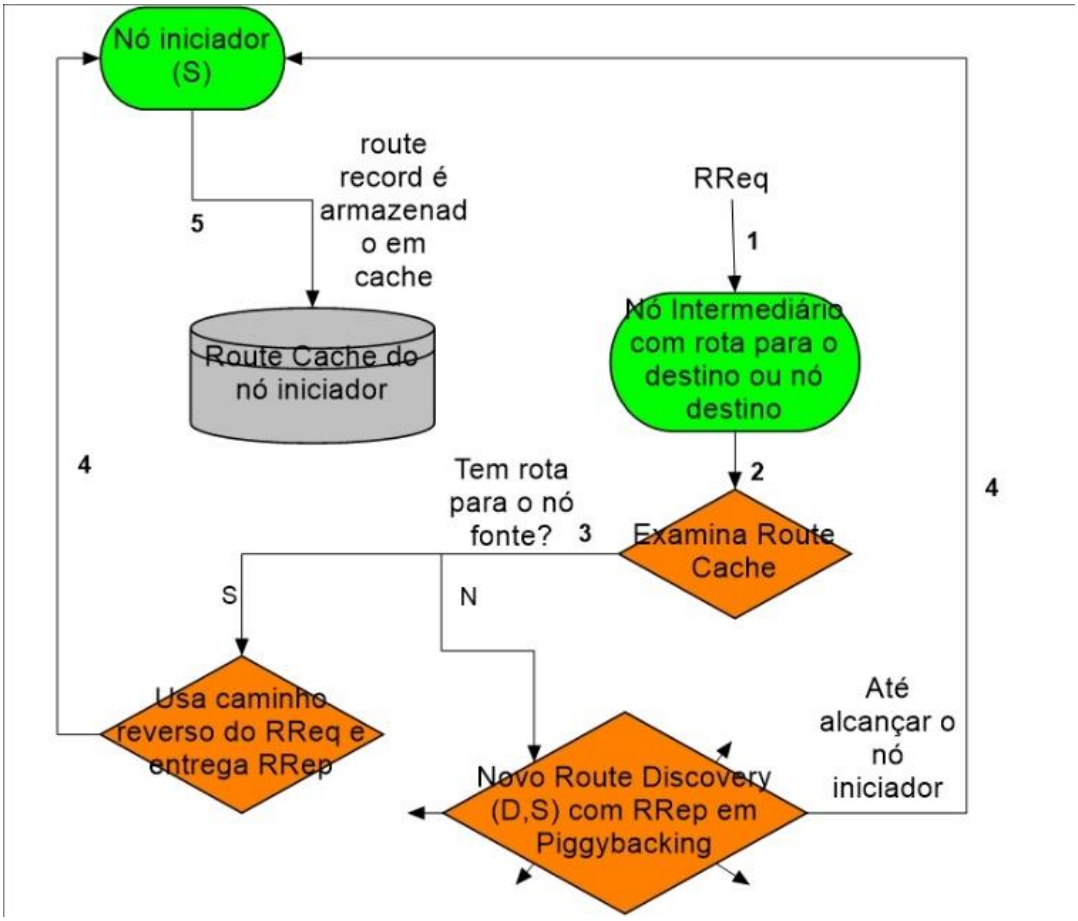

Figura 7 - Resumo do Route Reply

4.1.3. Considerações sobre o envio de pacotes

Para cada pacote a ser enviado por um nó, sempre é procedido um descobrimento de rotas, *Route Discovery*. Em determinados casos, podem ocorrer tentativas de envio de pacotes sem que exista efetivamente, durante um tempo, uma rota válida para esse processo, fenômeno a que se denomina particionamento[6] da rede (RFC 4728). Nesses casos, introduz-se o conceito do *Send Buffer* de um nó, que é uma fila de pacotes que não podem ser enviados por ele, por ainda não se conhecer rota adequada para tal.

Assim, feita a solicitação para descobrimento de rotas, enquanto o nó solicitante não recebe RRep referente à sua solicitação, o pacote é armazenado em seu *Send Buffer* para posterior envio, associando a ele um tempo de armazenamento. Esse tempo de armazenamento, referenciado aqui como *SendBufferTimeout*, tem como objetivo evitar que ocorra *overflow*[7] desse *buffer*.

Ainda com relação à solicitação de descobrimento de rotas, o protocolo prevê, devido à mobilidade da rede, a existência de *Route Requests* improdutivos que podem congestionar a rede (*overhead*). Devido a isso, é importante que o nó limite a quantidade de RReq realizadas para um mesmo destino, pois este pode estar temporariamente indisponível. Dessa forma, o algoritmo de recuo binário exponencial é utilizado para limitar a taxa de solicitações de rota para aquele mesmo destino, dobrando o valor de *timeout* para descobrimentos de rota sucessivos para o mesmo destino.

A Figura 8 apresenta o esquema de pacotes pendentes no *Send Buffer*:

[6] O particionamento da rede significa que, em determinado instante, não há uma sequência de nós através dos quais o pacote pode ser encaminhado para alcançar o destino

[7] Aqui significando um estouro da capacidade do buffer, ultrapassando seus limites de escrita.

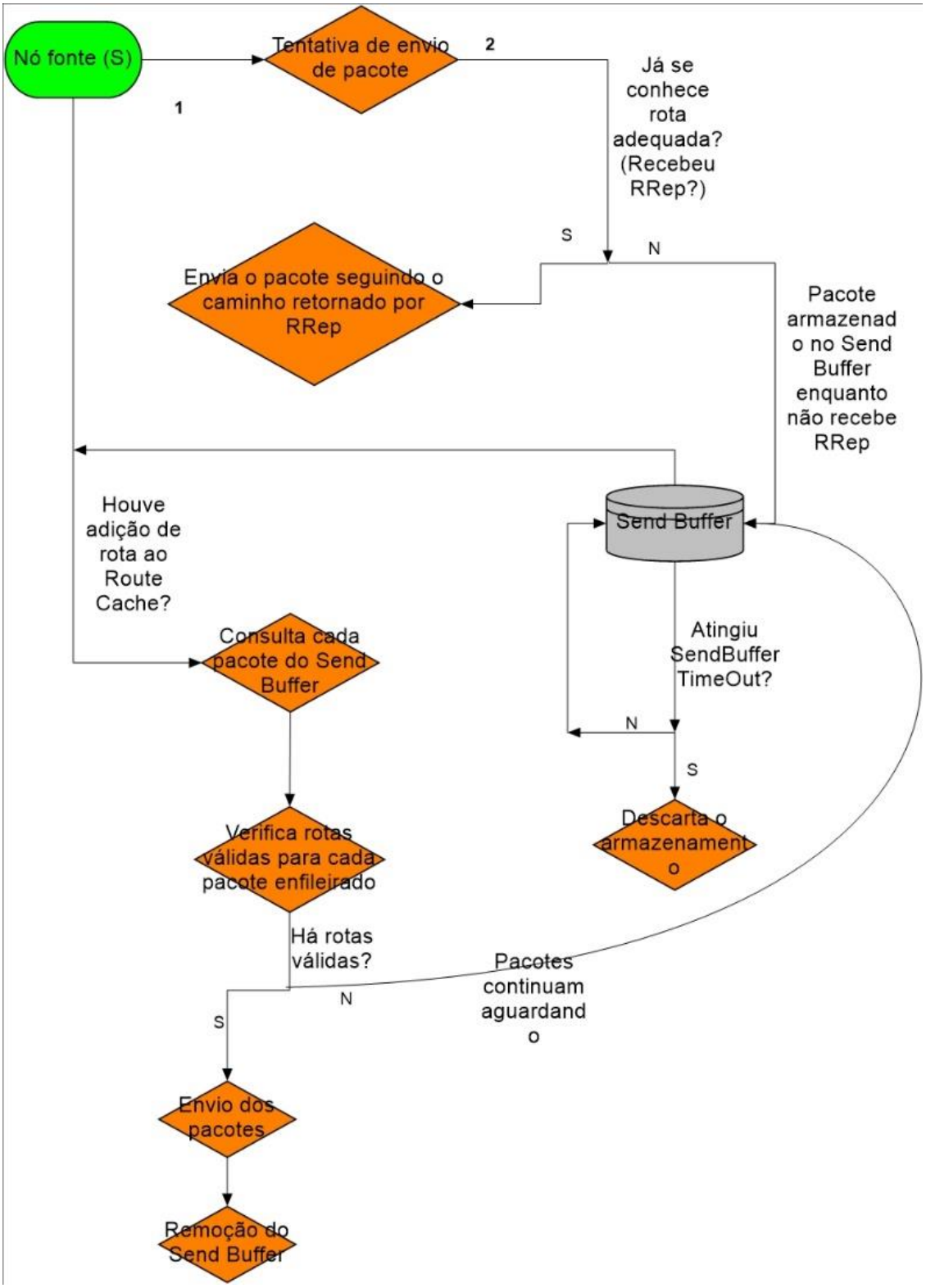

Figura 8 - Esquema de funcionamento para pacotes pendentes

4.2. Route Maintenance

O *Route Maintenance* é o mecanismo que permite a um nó detectar a ocorrência de alterações significativas na topologia da rede, enquanto usa uma rota para um destino, e que o impedem de continuar a propagação do pacote. Também envolve o conjunto de medidas realizadas para correção ou contorno dessas alterações. Vale ressaltar que a *Route Maintenance* se vale da cooperação entre os nós, ou seja, cada nó da rede é responsável por confirmar se dados podem ser trafegados por uma determinada rota até o próximo salto. Assim, a detecção de um link inoperante pode permitir que o nó fonte (S) consulte seu cache em busca de rotas alternativas para o destino (D). Caso não haja, procede-se novo *Route Discovery*, como mencionado anteriormente.

A confirmação de sucesso no envio e recebimento dos pacotes é feita por recebimento de *Acknowledgement* (ACK), que são mensagens de reconhecimento. Então, a fim de confirmar o recebimento de um pacote enviado, um nó pode solicitar ao nó de próximo salto um *Acknowledgement Request* (Req Ack). A resposta ao Req Ack pode ocorrer de três maneiras:

- em pacote separado;
- em *piggybacking* de retransmissão do pacote original, ou;
- em *piggybacking* de qualquer pacote que possua o mesmo próximo salto da solicitação de Req Ack.

Ainda, se o limite máximo para o número de Req Ack é atingido, o link para o próximo salto é considerado quebrado, procedendo-se à sua remoção como próximo salto na *Route Cache* e retornando *Route Error* (RErr) para cada nó que usou esse link pela última vez. Procedimento análogo é adotado quando um nó intermediário trafegando um pacote detecta que o link de próximo salto está quebrado. Nesse caso, além de proceder com o retorno de RErr, como citado anteriormente, o nó deve manipular de maneira similar qualquer pacote pendente que esteja enfileirado e destinado ao nó de próximo salto defeituoso. Especificamente, o nó consulta sua *Network Interface Queue*[8] e seu *Maintenance Buffer*[9]para encontrar pacotes nessas condições. Se houver, os pacotes são removidos dessas listas, é encaminhado um RErr desses pacotes para o nó fonte (S), e se houver rota alternativa para eles, deve-se proceder ao *Packet Salvaging* (ou salvamento de pacotes), caso contrário, é descartado. O nó fonte, então, utiliza rota alternativa, se houver em sua *Route Cache*, ou então inicia novo Route Discovery para o destino (D).

[8] È uma fila de pacotes da pilha do protocolo de rede aguardando para serem transmitidos pela interface de rede (RFC 4728). Essa fila é utilizada para reter pacotes enquanto a interface de rede está em processo de transmissão de outro pacote.

[9] O *Maintenance Buffer* de um nó consiste em uma fila de pacotes enviados por esse nó que estão aguardando a confirmação de alcance de próximo salto como parte do Route Maintenance (RFC 4728).

A figura abaixo traz o padrão do option para o *Route Error*:

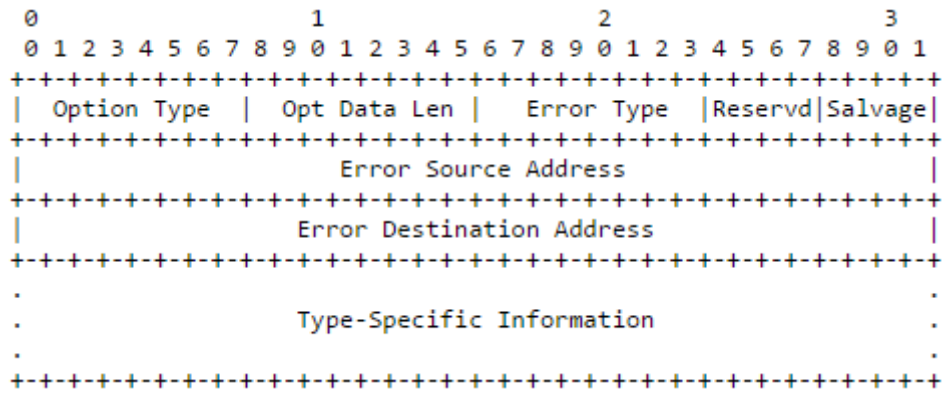

Figura 9 - *Option* padrão para o *Route Error*

Option Type: idem à explicação fornecida para o *Route Request*.

Opt Data Len: inteiro de 8 bits sem sinal. Comprimento do campo *option*, em octetos, excluindo os campos *Option Type* e *Opt Data Len*. Este campo deve ser setado para 10, acrescido do tamanho de qualquer informação de tipo específica presente no *Route Error*. Demais extensões ao formato do *option* do RErr devem também ser incluídas após as informações específicas de tipo acima citadas. A presença dessas extensões serão indicadas pelo campo *Opt Data Len*. Quando o *Opt Data Len* for maior do que o requerido para a porção fixa do *Route Error*, acrescida da informação específica de tipo, os octetos remanescentes são interpretados como extensões.

Error Type: é o tipo de erro encontrado. Os seguintes tipos são definidos:

1 = NODE_UNREACHABLE

2 = FLOW_STATE_NOT_SUPPORTED

3 = OPTION_NOT_SUPPORTED.

Reserved: deve ser enviado como 0 e ignorado na recepção.

Salvage: inteiro de 4 bits sem sinal. Copiado do campo *Salvage* no *option* do DSR *Source Route* do pacote que aciona o *Route Error*. O contador de salvamento total, total *salvage count*, é a soma de 1 mais o valor no campo *Salvage* do *option* da *Route Error*, para cada uma destas.

Error Source Address: é o endereço do nó que origina a *Route Error* (isto é, o nó que deveria trafegar o pacote adiante, mas descobriu erro no link).

Error Destination Address: endereço do nó para o qual o *Route Error* deve ser entregue. Por exemplo, quando o campo de *Error Type* é setado para NODE_UNREACHABLE, esse campo será setado para o endereço do nó que gerou a informação de roteamento reivindicando que o salto a partir do *Error Source Address* para o *Unreachable Node Address* (especificado na informação específica de tipo) era um salto válido.

Type-Specific Information: informação específica para o *Error Type* da mensagem de *Route Error* considerada.

A figura a seguir traz exemplo do *option* para o *Acknowledgement Request*:

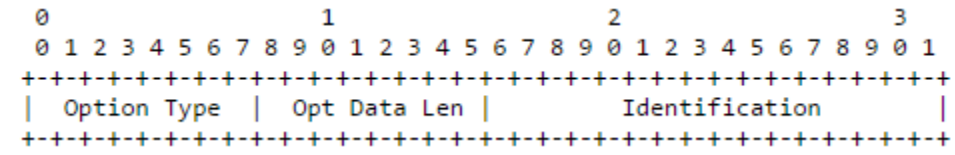

Figura 10 - Cabeçalho padrão para o *Acknowledgement Request*

Option Type: análogo aos anteriores. Entretanto, nós que não compreenderem este campo irão remover a opção e retornar *Route Error*.

Opt Data Len: inteiro de 8 bits sem sinal. Comprimento do *option*, em octetos, excluindo os campos *Option Type* e *Opt Data Len*.

Identification: é setado para um único valor e é copiado no campo *Identification* do *Acknowledgement* , quando retornado pelo nó que recebe o pacote ao longo deste salto.

Abaixo encontra-se o padrão do option para o *Acknowledgement*:

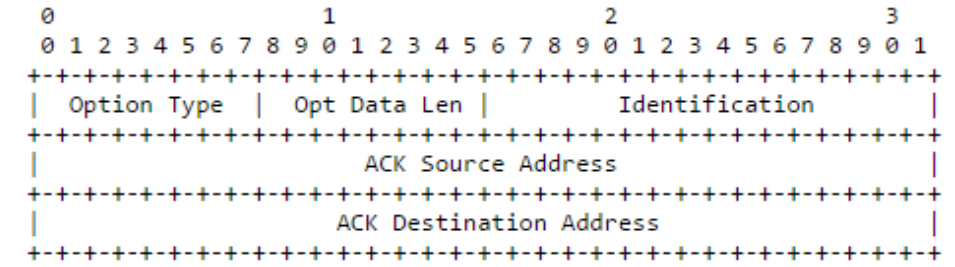

Figura 11 - Cabeçalho padrão do *Acknowledgement*

Option Type: análogo aos anteriores. Nós que não compreendam essa opção irão remover a opção.

Opt Data Len: inteiro de 8 bits sem sinal. Comprimento do *option*, em octetos, excluindo os campos *Option Type* e *Opt Data Len*.

I**dentification**: copiado do campo *Identification* do *option* do *Acknowledgement Request* para o pacote que está sendo confirmado.

ACK Source Address: endereço do nó que origina o ACK.

ACK Destination Address: endereço do nó para o qual o ACK deverá ser entregue.

O *Route Maintenance* é resumido na Figura 13:

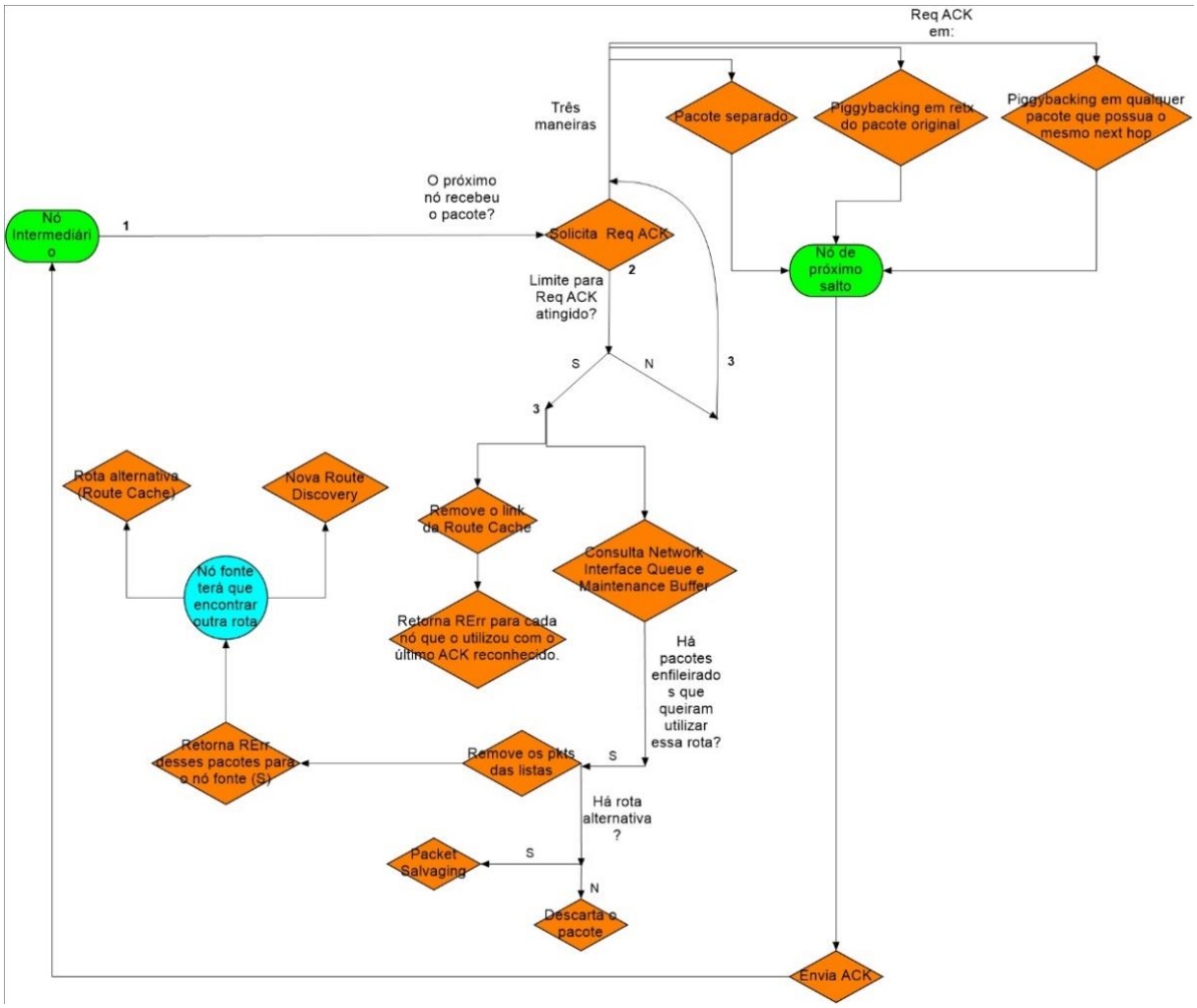

Figura 12 - Esquema de funcionamento do Route Maintenance

4.3. Route Cache

O *Route Cache* é uma lista que cada nó possui onde são guardadas informações de rotas, devendo ser capaz de armazenar mais de uma rota para cada destino. Um nó que trafegue um pacote adiante, ou que escute por acaso algumas transmissões de outros pacotes, deve adicionar todas as informações referentes a rotas desses pacotes em sua própria *Route Cache*, sendo este armazenamento válido também para a remoção de rotas (o que, por exemplo, pode ser aprendido de RErr que por ali trafeguem). Para a escolha de rotas, são adotados critérios, como um algoritmo de menor distância.

A cada nova entrada em uma *Route Cache* é associado um timeout associado, aqui referenciado como *RouteCacheTimeout*, visando-se evitar o *overflow* do cache. Se esse limite é atingido, procede-se ao gerenciamento de armazenamento dessas rotas, em que é adotado um critério para a remoção das já existentes e inserção de novas rotas, podendo-se excluir as menos utilizadas ou adotar outro critério. Também, após

cada nova adição de rotas, se faz necessário que o nó consulte cada pacote presente em seu *Send Buffer*, para verificar a existência de rotas válidas para os pacotes enfileirados. Caso haja, procede-se o envio do pacotes e, logo após, sua remoção do *Send Buffer*.

O cache pode ser realizado de duas formas: link cache ou path cache. No *link cache,* cada link individual (salto), retornados dos pacotes do RRep ou de outros recursos, é adicionado a uma estrutura unificada de grafos de dados como uma visão da topologia da rede pelo nó em foco. A consulta a uma rota é feita, então, por meio de consultas mais complexas a algoritmos[10] de buscas em grafos, para encontrar o melhor caminho em direção ao nó destino (D). O *link cache* é mais robusto, pois utiliza efetivamente todo o potencial de informação que um nó pode fornecer sobre o estado da rede, uma vez que link caches descobertos de diferentes *Route Discoveries* podem ser acoplados para constituir novas rotas.

Já no *path cache*, a rota retornada em cada RRep recebida pelo iniciador (S) representa um caminho completo (uma sequência de links, ou saltos) encaminhando para o nó destino. O armazenamento de cada caminho separadamente no *Route Cache*, constitui o processo de *path cache*. Sua implementação é simples e garante que todas as rotas estejam livres da ocorrência de *loops*. A consulta a uma determinada rota na *Route Cache* estruturada por *path cache* é feita procurando-se qualquer caminho, ou prefixo deste, que o direcionem ao destino. Entretanto, no *path cache*, não é possível utilizar-se o acoplamento de rotas como permite o *link cache*, devido à separação de cada caminho (*path*) individual no cache.

[10] A RFC 4728 cita o algoritmo do caminho mais curto de Dijkstra como um dos utilizados

5. IMPLEMENTAÇÃO DO DSR

Para a implementação do protocolo, optou-se pela linguagem C++, por fornecer a integração com o Linux desejável. Além disso era a linguagem comum a todos o membros da equipe.

Quanto a organização do programa, para cada tipo de opção foi implementada uma classe correspondente. As classes relativas a opções do Route Request herdavam de uma classe pai virtual chamada Message e as mensagens relativas ao Route Maintenance herdavam de uma classe pai virtual chamada MessageMaintenance. Essa estratégia foi adotada para que fosse possível usar-se do polimorfismo. Ou seja, criou-se um ponteiro para a classe pai e conforme seja de interesse do programa, atribui-se a ele uma referência de um objeto de classe filha. A partir daí o ponteiro da classe pai assume as características da classe filha, permitindo usá-lo em trechos do código em que não se sabe de antemão que tipo de objeto deverá ser trabalhado. Outra alternativa seria declarar esses objetos conforme fossem necessários, porém a desvantagem é que eles estariam restritos ao escopo em que foram declarados. Segue um trecho de código com essa idéia.

```
Message *ptr;       //declaração do ponteiro para classe abstrata
RReq objRReq;                       //declaração do objeto da classe correspondente a uma opção DSR

if(bufferRecepcao[0] == 1){         //     identificação do tipo de      mensagem recebida
   ptr = & objRReq;                 //     ptr agora tem as características de um RReq
   ptr ->lerBuffer(bufferRecepcao); //     funçao específica para o RReq que associa a informação
                                    //de cada byte a um atributo de RReq de acordo com o cabeçalho
                                    //presente na RFC.

   //continua...
```

Figura 13 - Exemplo de código aplicando o polimorfismo à leitura do buffer de entrada

Ao se colocar cada opção DSR em uma classe distinta proporcionou ao programador diversas vantagens, como por exemplo, ter métodos de leitura/escrita de buffer de entrada específico para tipo de mensagem, bem como facilita a abstrair os diferentes tipos em cada situação do protocolo. Como pôde ser visto na Figura 14, o primeiro byte de cada opção sempre é o tipo, um número único que identifica qual das opções está presente naquela sequência de caracteres(buffer). Assim a estratégia geral no tratamento de uma mensagem recebida é ler esse primeiro byte da sequência. Lido esse byte já se sabe o teor da mensagem e o objeto da classe abstrata pode receber o tipo certo de referência para classe correspondente. Assim pode-se usar função de leitura já específica para aquele tipo, como pode ser visto na figura acima.

Para garantir conectividade com a rede, fez-se uso de dois tipos de socket, cada um implementado em uma classe. Ao contrário de uma rede infraestruturada convencional com um serviço de acesso a uma página *web,* por exemplo, estão bem definidos os lados cliente e servidor. No caso de um nó operando em uma rede Ad Hoc, esses dois lados não são definidos. Todo nó hora é cliente, hora é servidor. Por isso foram implementadas duas classes, uma para o comportamento cliente e outra para o comportamento servidor.

O tipo de socket não é explicito no texto da RFC, porém para atendê-la, deve-se implementar o serviço com sockets do tipo raw. Esse tipo de socket permite ter acesso ao cabeçalho IP do pacote. Um exemplo desta necessidade pode ser encontrado na sessão (5.1.1) que trata da opção Route Request. Ela é retransmitida até chegar no .nó marcado como Target. O detalhe é que a cada retransmissão o campo Source address do cabeçalho IP deve ser mantido constante, ou seja deve ser salvo em uma variável do programa e após o processamento do cabeçalho DSR e de suas opções o cabeçalho IP da mensagem a ser retransmitida é configurado com o valor salvo na variável do programa, garantindo, assim, a constância desse campo, para essa situação.

Outro detalhe relevante está nas estruturas que envolvem temporização. Por exemplo, na sessão (5.1.1) foi explicada estrutura Route Request Table, que tinha o objetivo evitar que um nó processasse a mesma Route Request indefinidas vezes. Essa estrutura foi implementada da seguinte forma: implementou-se uma classe chamada RReqTable cujo principal atributo privado é um vetor (template vector do C++) cujos elementos são objetos da classe Route Request. O nó ao transmitir uma mensagem com Route Request, irá adicionar a esse vetor o objeto correspondente que ficará armazenado na lista por um período de tempo pré-definido pois o método que realiza o *append* do novo objeto, no final chama um processo (*thread*) que ao final de uma temporização automaticamente apaga ele do vetor.

Para que fosse possível ter um bom nível de abstração na hora de ter contato com a interface de rede, para criar ou destruir rotas, foi implementada uma classe chamada Bash que implementava métodos com essas funções.

Para fins de auxílio no debug das funções, foi implementado um programa que gera todos os tipos de mensagens do protocolo. Esse programa fará as vezes do iniciador nos testes realizados. O protocolo funciona colocando todos os pacotes a serem transmitidos em um buffer e os envia de acordo com as trocas já explicadas de mensagens. Essa parte não foi implementada, todavia o comportamento da rede é exatamente o mesmo com as mensagens geradas pelo programa iniciador.

6. TESTES COM A IMPLEMENTAÇÃO

O principal objetivo na parte da simulação é validar o funcionamento das estruturas lógicas do protocolo, procurando as falhas mais latentes. Também buscou-se analisar os efeitos das mensagens de controle no tráfego de informações. Essa seção será dividida em cenários de teste, explicando os objetivos e resultados de cada um.

6.1. Teste de Propagação de Route Request

O objetivo desse teste é demonstrar a propagação das mensagens de Route Request até chegar-se no nó marcado como Target . Feito isso, o nó Target retribuirá com uma mensagem Route Reply que ao ser recebida será processada e atualizará a tabela de Cache do nó originado. Para isso considere a configuração a seguir:

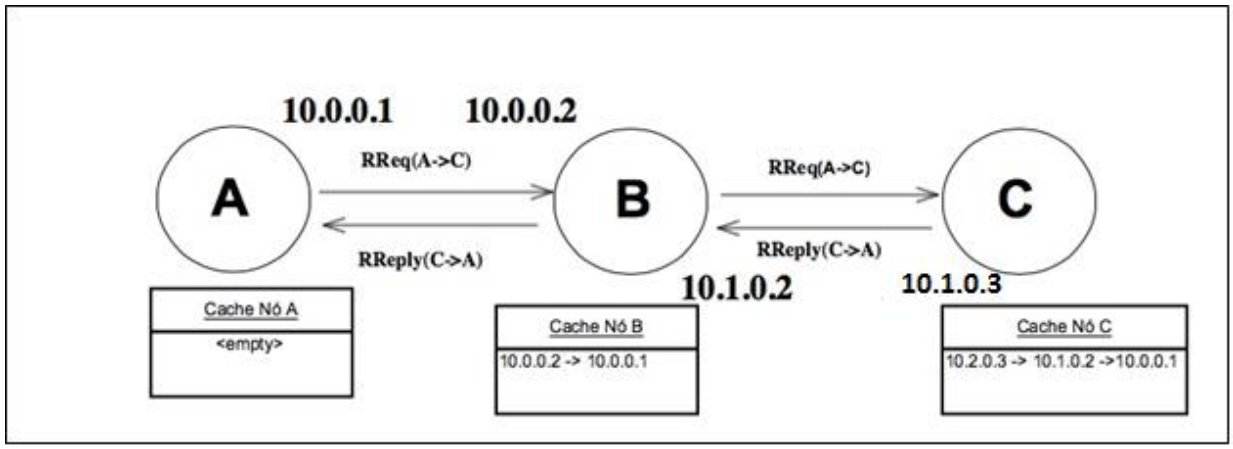

Figura 14 - Teste de Propagação do Route Request

Nesse teste é possível perceber que A recebe com sucesso o Route Reply de C e incrementa seu cache.

6.2. Teste de Cached Route Reply

O objetivo desse teste é demonstrar a Cached Route Reply, que é a resposta que um nó intermediário dá para o iniciador da Route Request quando ele tem rota para o Target, conforme figura a seguir:

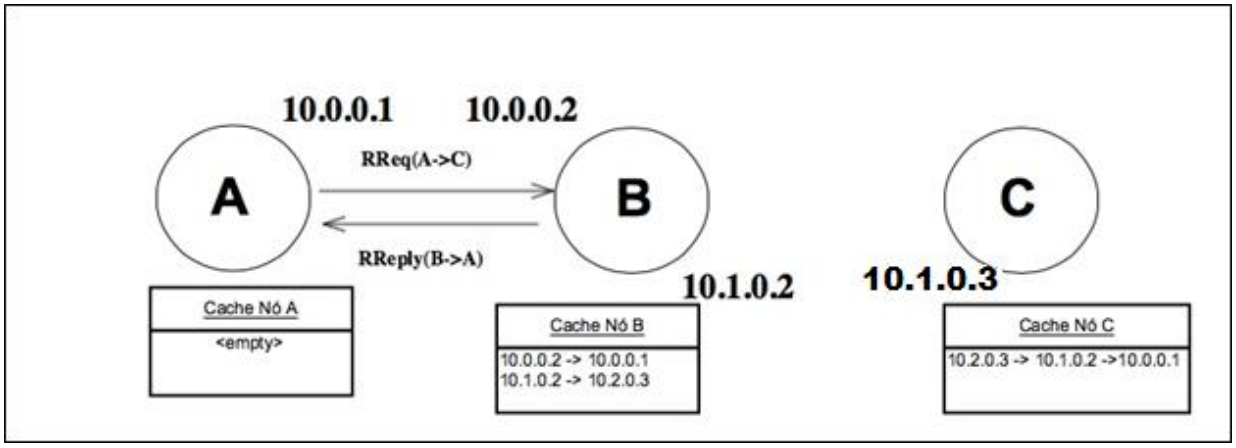

Figura 15 - Teste de Cached Route Reply

Neste caso, A quer transmitir para C, mas não tem rota para ele em seu Cache. Ao iniciar o Route Discovery, com um Route Request de A para B, ao receber essa mensagem B verifica que tem rota para C. Por isso, B não reencaminha o Route Request, ele devolve para A um Route Reply com a rota de seu Cache.

6.3. Teste Route Maintenance

O objetivo desse teste é demonstrar a capacidade de trafegar um pacote com um dado arbitrário, usando uma opção Source Routing. Neste cenário A quer transmitir o pacote para D. O Cache de A já possui rota para D, então não há necessidade de iniciar uma Route Discovery. Como na transmissão da mensagem com o Source Route, cada nó é responsável por confirmar a disponibilidade do próximo nó, antes de encaminhá-lo, há essa troca de ReqAck's e Ack's até o dado(linha vermelha na figura a seguir chegar ao destino).

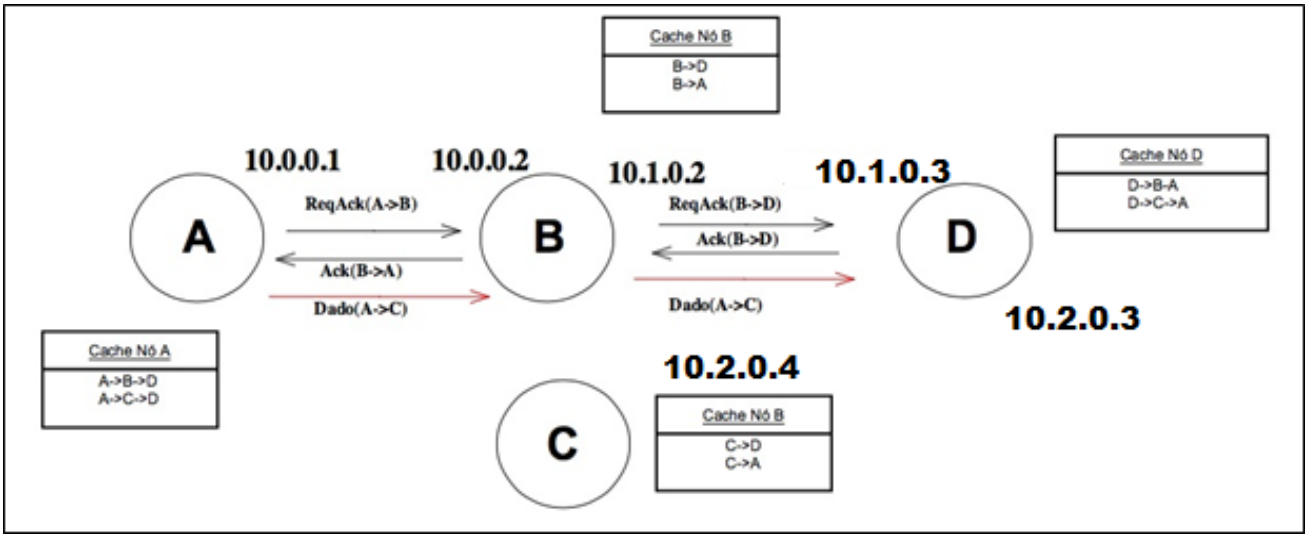

Figura 16 - Teste do Route Maintenance

Note que apesar de haver rota envolvendo C, em momento algum ele entra no processo, graças à opção Source Route.

6.4. Teste de Identificação de Link Quebrado

O objetivo desse teste é comprovar a eficácia do Route Maintenance para detectar um link quebrado e mostrar alguma das possíveis soluções. Observe a figura a seguir:

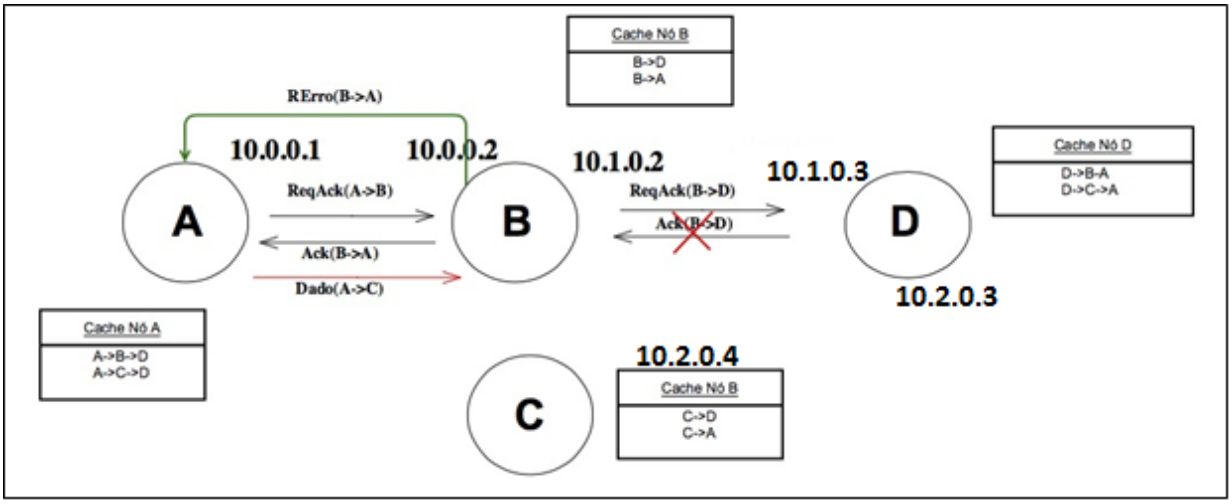

Figura 17 - Teste do Link Quebrado

Nesse cenário o nó D não respondeu por estar *offline* (propositadamente desconectado). Ao perceber isso, ou seja quando der *timeout* dos ReqAck e o número de retransmissões chegar no seu máximo, o nó B enviará uma mensagem de erro do

tipo Nó Indisponível(*Node-Unreachable*) para o originador A. A poderá optar por outra rota em seu Cache, por exemplo via C, ou poderia iniciar uma Route Discovery.

6.5. Teste de Propagação em Vazio

O teste final compreende 4 máquinas operando em linha sem nenhuma rota em seus Caches. O objetivo é estabelecer a rede de modo a transmitir de A para D. Observe a figura a seguir:

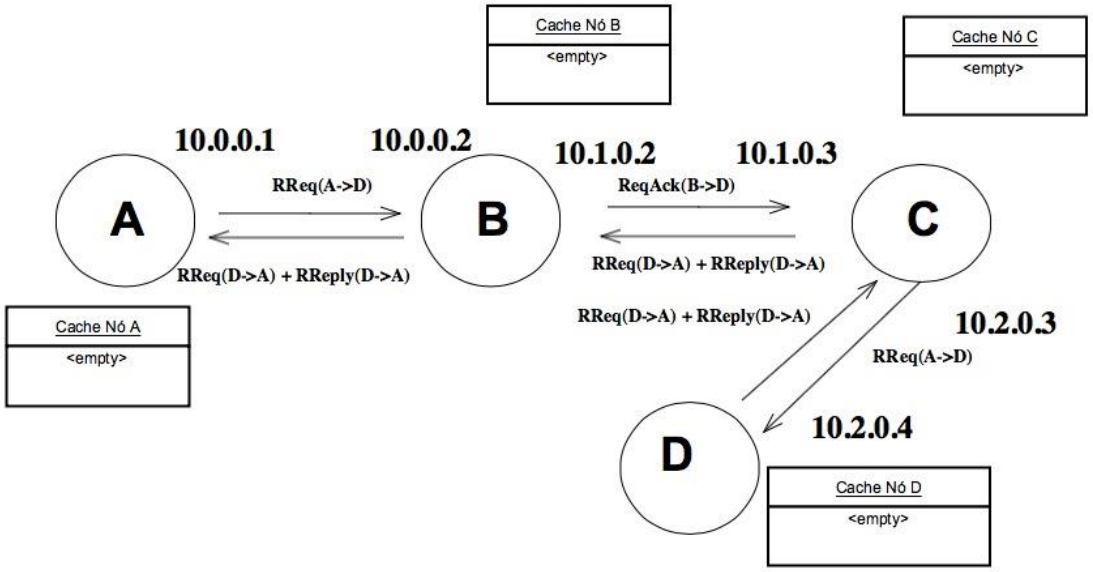

Figura 18 - Teste em Vazio

Note que como D não conhece rota para A, sua resposta Route Reply vem acompanhada de um Route Request em *piggyback.*

7. CONCLUSÃO

Este trabalho buscou apresentar conceitos de redes sem fio com foco nas redes ad hoc. Aqui foram abordadas noções deste tipo de topologia, apontando problemas e descrevendo os principais protocolos. Após a análise das características destes protocolos, decidiu-se pela implementação do Dynamic Source Routing.

A implementação do protocolo foi feita na linguagem C++ e se mostrou funcional para diversos tipos de topologia. Foram implementados todos os aspectos do DSR, excetuando-se os opcionais de *Flow State*.

O desenvolvimento apresentou resultados satisfatórios. Foram realizados testes para verificar o funcionamento da descoberta, da manutenção e armazenamento de rotas, utilizando-se de troca de mensagens simples. Os testes mostraram que o protocolo funciona bem em redes com poucos nós. Assim como demonstraram a efecácia do Route Dicovery na obtenção de rotas entre dois nós e do Route Maintenace na preservação das rotas.

8. PRÓXIMOS PASSOS

Para etapas futuras, com relação a implementação, sugere-se que o programa que implementa o protocolo tenha algum tipo de ligação com a interface de rede que permita que todos os pacotes passem antes pelo programa. Assim seria possível implementar o Send Buffer e a rede se tornaria de fato útil para trafego de dados.

Outro aspecto a ser considerado é relativo a atribuição de endereços das interfaces e subinterfaces dos nós implementando o protocolo. Um estudo aprofundado de endereçamento móvel, como o Mobile Ip, por exemplo são candidatos promissores a uma solução nesse quesito.

Finalmente, melhoras pontuais de desempenho podem ser atingidas implementando algoritmos que dentro da tabela de Cache de um dado nó, retorne uma determinada rota de maneira otimizada, considerando eventualmente algum fator de ponderação extraido da própria rede.

9. REFERÊNCIAS BIBLIOGRÁFICAS

COMER, D.E., Interligação em Redes com TCP/IP: Princípios, protocolos e arquitetura. Vol 1. 3 Ed. Rio de Janeiro: Editora Campus, 1998. 672p.

COMER, D.E., Interligação em Redes com TCP/IP:Projeto, implementação e detalhes internos. Vol II. 3 Ed. Rio de Janeiro: Editora Campus, 1998. 592p.

DEITEL,H.M.;DEITEL,P.J. *C++ How to Program*. 4 Ed. New Jersey: Prendice Hall, 2003. 1321p.

FOROUZAN, Behrouz A.Data Communications and Networking. 4.ed. Nova Iorque: McGraw-Hill, 2007.1134p. (McGraw-Hill Forouzan networking series)

JOHNSON, D.; HU, Y.; MALTZ,D. The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4. Disponível em: <https://www.ietf.org/rfc/rfc4728.txt> . Acesso em 22 mai. 2015.

JARGAS, A.M., Shell Script Profissional. São Paulo: Novatec Editora, 2008. 480p.

KUROSE,J.F., *Redes de Computadores e a Internet*: Uma Abordagem Top-Down. 3 Ed. São Paulo: Pearson Addison Wesley, 2006. 634p.

MORAES, Ana Luiza D.; XAUD, Arthur F. dos Santos; XAUD, Marco F. dos Santos. Redes Ad Hoc. Protocolos DSR, AODV, OLSR e DSDV. Disponível em: <http://www.gta.ufrj.br/grad/09_1/versao-final/adhoc/index.html>. Acesso em 24 mai 2015.

D. JOHNSON, Y. HU e D. MALTZ. RFC 4728: The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4, Fevereiro de 2007

TANENBAUM, Andrew S;WETHERALL, David J. Computer Networks. 4.ed. Boston, Massachussets: Prentice Hall, 2002. 933p.

WELCH, Jennifer. CPSC 689: Discrete Algorithms for Mobile and Wireless Systems. Disponível em: https://parasol.tamu.edu/~welch/. Acesso em 24 mai 2015.

Y.1541, ITU-T. Network performance objectives for IP-based services. Disponível em: https://www.itu.int/rec/T-REC-Y.1541-201112-I/en. Acesso em 23 mai 2015.

10. ANEXOS

10.1. Código do DSR Implementado

```
//
//  main.cpp
//  Route Request
//
//  Created by Pedro Loami on 01/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include <iostream>

#include "ServerSocketUDP.h"

#include "SocketRaw.h"
#include "RReqTable.h"
#include "Context.h"
#include "Bash.h"
#include "ackTable.h"

#include "Interface.h"
#define PORTA 4950

using namespace std;

int main(int argc, const char * argv[]) {
    // **************
    // *   TESTES   *
    // **************

    //-------- Example Addresses -------

    //_____________________________________


    //x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~[ Enviromment Setup
]~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x

    //----------Interface Setup---------

    Interface _inter;
    cout<<_inter<<endl;
```

```
//_____________________________________

//------------ Seed RReq -----------
//      Seed for RReq's identifiers
//from this node.
uint16_t gemRReq = 0;
//_____________________________________

//------------ Seed SR ------------
//      Seed for Source Route's iden-
// tifiers from this node.
uint16_t gemSR = 0;
//_____________________________________

//-------- Example Addresses -------

Ipv4 ip1(10,0,0,1), ip2(10,0,0,2),ip3(10,1,0,3),ip4(10,0,0,4),ip5(10,0,0,5);
//Ipv4 ip1(192,168,0,5), ip2(192,168,0,2),ip3(192,168,0,3),ip4(192,168,0,4);

//_____________________________________


//--------- Intermed Example --------

std::vector<Ipv4> vecIp1,vecIp2,vecEmpty;  //intermeds

vecIp1.push_back(ip2);
vecIp2.push_back(ip3);
vecIp2.push_back(ip4);

//_____________________________________

//---------- Route Example --------

Route rota1,rota2;

rota1.setRoute(ip3,vecIp1);
rota2.setRoute(ip3,vecEmpty);
//_____________________________________

//------------ RReq Table ----------

RReqTable tableRReq;
//_____________________________________
```

```
    //------------ Ack Table ----------

    AckTable tableAck;

    //_____________________________________

    //------------ Cache Table ----------

    Cache _cache;

    //_cache.appendCache(rota1);
    //_____________________________________



    //x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~[ Inicialization Setup
]~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x

    int operacao = 1;
    Context contexto;
    ServerSocketUDP server(PORTA);
    unsigned char * inputBuffer;


    //x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~[ Testes Voids
]~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x
    //RReqTableType rreqTableTest1,rreqTableTest2;
    //rreqTableTest1.setIdNumber(0, 1);
    //rreqTableTest1.setTarget(ip3);

    //tableRReq.appendRReq(rreqTableTest1);

    //rreqTableTest2.setIdNumber(0, 2);
    //rreqTableTest2.setTarget(ip1);

    //tableRReq.appendRReq(rreqTableTest2);

    //x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~[ Program`s Loop
]~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x~x

    while(operacao){
        std::cout<<"\n----------------\nEntrei no laço!\n----------------\n"<<endl;
```

```
        _cache.printCache();
        inputBuffer = server.receber(1000);
        contexto.processInput(inputBuffer, _cache, server.getSender(), tableRReq,tableAck,
gemRReq,gemSR);
        std::cout<<"\n-----------------\nSaí do laço!\n-----------------\n"<<endl;

    }//while


    std::cout<<"FIM DO main.cpp"<<std::endl;
    return 0;
}
//
//  AckOpt.h
//  Cliente UDP
//
//  Created by Pedro Loami on 11/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Cliente_UDP__AckOpt__
#define __Cliente_UDP__AckOpt__

#include "MessageMaintenance.h"
#include <iostream>

class AckOpt{
    uint8_t dataLen;
    uint8_t idt[2];

    Ipv4 ackSourceAdd;
    Ipv4 ackDestAdd;
public:
    AckOpt();

    char *bufferSaida();
    void lerBuffer(unsigned char *buffer);


    uint8_t getType() const;

    uint8_t getDataLen() const;
    void setDataLen(uint8_t);
```

```
    uint8_t * getIdt() const;
    void setIdt(uint16_t &);
    //void setIdt(uint8_t *);

    Ipv4 getAckSourceAdd() const;
    void setAckSourceAdd(Ipv4);

    Ipv4 getAckDestAdd() const;
    void setAckDestAdd(Ipv4);

    friend std::ostream &operator<<(std::ostream &, const AckOpt &);
    bool operator == (const AckOpt &) const;
    bool operator != (const AckOpt & right) const{
        return !(*this == right);
    }

};
#endif /* defined(__Cliente_UDP__AckOpt__) */
//
//  AckOpt.cpp
//  Cliente UDP
//
//  Created by Pedro Loami on 11/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "AckOpt.h"

AckOpt::AckOpt(){

    this->dataLen =0x0A;
    uint16_t id = 0x0000;
    this->setIdt(id);
    this->ackSourceAdd.setIp(0, 0, 0, 0);
    this->ackDestAdd.setIp(0,0,0,0);
}

char *AckOpt::bufferSaida(){
    char * buffer;
    buffer = new char [(int)dataLen+2];

    buffer[0] = this->getType();

    buffer[1] = this->getDataLen();
```

```
    buffer[2] = this->getIdt()[0];
    buffer[3] = this->getIdt()[1];

    unsigned char *aux1 = this->ackSourceAdd.getIp();
    buffer[4] = aux1 [0];
    buffer[5] = aux1 [1];
    buffer[6] = aux1 [2];
    buffer[7] = aux1 [3];

    unsigned char *aux2 = this->ackDestAdd.getIp();
    buffer[8] = aux2 [0];
    buffer[9] = aux2 [1];
    buffer[10] = aux2 [2];
    buffer[11] = aux2 [3];

    return buffer;
}

void AckOpt::lerBuffer(unsigned char *buffer){
    if (buffer[0] != 32) {
        throw "ERROR! This is not an Acknowledgement DSR message.";
    }
    this->dataLen = buffer[1];
    this->idt[0] = buffer[2];
    this->idt[0] = buffer[2];

    this->ackSourceAdd.setIp(buffer[4], buffer[5], buffer[6], buffer[7]);

    this->ackDestAdd.setIp(buffer[8], buffer[9], buffer[10], buffer[11]);
}
uint8_t AckOpt::getType() const{
    return 32;
}

uint8_t AckOpt::getDataLen() const{
    return this->dataLen;
}

void AckOpt::setDataLen(uint8_t _dataLen){
    this->dataLen = _dataLen;
}

void AckOpt::setIdt(u_int16_t &_gem){
    this->idt[1] = _gem & 0xFF;
```

```
    this->idt[0] = _gem >> 8;
    //_gem++;
}

uint8_t * AckOpt::getIdt()const{
    static uint8_t * aux;
    aux = new uint8_t[2]();
    aux[0] = this->idt[0];
    aux[1] = this->idt[1];

    return aux;
}

//void AckOpt::setIdt(uint8_t *input){
//    idt[0] = input[0];
//    idt[1] = input[1];
//
//}

Ipv4 AckOpt::getAckSourceAdd() const{
    return this->ackSourceAdd;
}

void AckOpt::setAckSourceAdd(Ipv4 _ip){
    this->ackSourceAdd = _ip;
}

Ipv4 AckOpt::getAckDestAdd() const{
    return this->ackDestAdd;
}

void AckOpt::setAckDestAdd(Ipv4 _ip){
    this->ackDestAdd = _ip;
}

std::ostream &operator<<(std::ostream &os, const AckOpt &objAckOptRef){
    os<<"Opt Type: "<<(unsigned int)objAckOptRef.getType()<<" <Ack Opt>"<<std::endl;

    os<<"dataLen: "<<(unsigned int)objAckOptRef.dataLen<<std::endl;

    os<<"Identification: "<<256*(unsigned int) objAckOptRef.idt[0]+(unsigned
int)objAckOptRef.idt[1]<<std::endl;

    os<<"Ack Source Add.: "<<objAckOptRef.ackSourceAdd<<std::endl;
```

```
    os<<"Ack Destination Add.: "<<objAckOptRef.ackDestAdd<<std::endl;

    return os;
}
bool AckOpt::operator == (const AckOpt &right) const{
    bool aux = true;
    std::cout<<"Right:\n"<<right<<std::endl;
    std::cout<<"this:\n"<<*this<<std::endl;

    if(this->dataLen != right.dataLen){
        aux =false;
    }
    if(this->idt[0] != right.idt[0]){
        aux =false;
    }
    if(this->idt[1] != right.idt[1]){
        aux =false;
    }
    if(this->ackSourceAdd != right.ackSourceAdd){
        aux =false;
    }
    if(this->ackDestAdd != right.ackDestAdd){
        aux =false;
    }
    return aux;
}

//
//  AckReqOpt.h
//  Cliente UDP
//
//  Created by Pedro Loami on 11/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Cliente_UDP__AckReqOpt__
#define __Cliente_UDP__AckReqOpt__

#include "MessageMaintenance.h"

class AckReqOpt{
    uint8_t dataLen;
    uint8_t idt[2];
```

```
public:
    AckReqOpt();

    char *bufferSaida();
    void lerBuffer(unsigned char *buffer);

    uint8_t getType() const;

    uint8_t getDataLen() const;

    uint8_t * getIdt()const;
    uint16_t  getNumericIdt()const;

    void setIdt(uint8_t,uint8_t);
    void setIdt(uint16_t &);

    friend std::ostream &operator<<(std::ostream &, const AckReqOpt &);

    bool operator == (const AckReqOpt &) const;
    bool operator != (const AckReqOpt & right) const{
        return !(*this == right);
    }


};

#endif /* defined(__Cliente_UDP__AckReqOpt__) */
//
//  AckReqOpt.cpp
//  Cliente UDP
//
//  Created by Pedro Loami on 11/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "AckReqOpt.h"

AckReqOpt::AckReqOpt(){
    uint16_t aux = 0x0000;
    this->setIdt(aux);
    this->dataLen = 2;
}
```

```
char *AckReqOpt::bufferSaida(){
    char * buffer;
    buffer = new char [(int)dataLen+2];

    buffer[0] = this->getType();

    buffer[1] = this->getDataLen();
    buffer[2] = this->getIdt()[0];
    buffer[3] = this->getIdt()[1];

    return buffer;
}

void AckReqOpt::lerBuffer(unsigned char *buffer){
    if (buffer[0] != this->getType()) {
        throw "ERROR! This is not an Acknowledgement Request DSR message.";
    }
    this->dataLen = buffer[1];
    this->idt[0] = buffer[2];
    this->idt[0] = buffer[2];

}

uint8_t AckReqOpt::getType() const{
    return 160;
}


uint8_t AckReqOpt::getDataLen() const{
    return this->dataLen;
}

uint8_t * AckReqOpt::getIdt()const{
    static uint8_t * aux;
    aux = new uint8_t[2]();
    aux[0] = this->idt[0];
    aux[1] = this->idt[1];

    return aux;
}
uint16_t AckReqOpt::getNumericIdt()const{
    return idt[0]<<8 | idt[1];
}
void AckReqOpt::setIdt(uint16_t &_gem){
```

```
    this->idt[1] = _gem & 0xFF;
    this->idt[0] = _gem >> 8;
    _gem++;
}

void AckReqOpt::setIdt(uint8_t _id1, uint8_t _id2){
    this->idt[0] = _id2;
    this->idt[1] = _id1;
}

std::ostream &operator<<(std::ostream &os, const AckReqOpt &objAckOptRef){
    os<<"Opt Type: "<<(unsigned int)objAckOptRef.getType()<<" <Ack Request
Opt>"<<std::endl;

    os<<"dataLen: "<<(unsigned int)objAckOptRef.dataLen<<std::endl;

    os<<"Identification: "<<256*(unsigned int) objAckOptRef.idt[0]+(unsigned
int)objAckOptRef.idt[1]<<std::endl;


    return os;
}

bool AckReqOpt::operator==(const AckReqOpt &right) const{
    bool aux = true;

    aux = aux && (this->dataLen == right.dataLen);
    aux = aux && (this->idt == right.idt);

    return aux;
}//
//  AckTable.h
//  Route Request
//
//  Created by Pedro Loami on 18/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__AckTable__
#define __Route_Request__AckTable__

#include "AckOpt.h"
#include "AckReqOpt.h"
#include "DSR_LIBRARY.h"
```

```
#include <thread>

class AckTable{
    std::vector<AckOpt> ackTableVector;

public:
    void appendAckReq(AckOpt _ackReq);

    bool isThere(AckOpt _ackReq);

    //bool isValid(RReqTableType _rreType);

    friend std::ostream &operator<<(std::ostream &, const AckTable &);

    bool operator == (const AckTable &) const;
    bool operator != (const AckTable & right) const{
        return !(*this == right);
    }

    static void deleteEntry(AckTable *, std::vector<AckOpt>::iterator);

};
#endif /* defined(__Route_Request__AckTable__) */
//
//  AckTable.cpp
//  Route Request
//
//  Created by Pedro Loami on 18/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "AckTable.h"


void AckTable::appendAckReq(AckOpt _ackReq){
    if( !isThere(_ackReq)){
        this->ackTableVector.push_back(_ackReq);
        std::cout << "Inserting ack " << _ackReq<< " in table." << std::endl;
        std::thread (deleteEntry, this, ackTableVector.begin()).detach();
    }

}

bool AckTable::isThere(AckOpt _ackReq){
```

```
    std::vector<AckOpt>::iterator it;
    it = find (this->ackTableVector.begin(),this->ackTableVector.end(),_ackReq);
    if(it != ackTableVector.end()){
        return true;
    }
    else{
        return false;
    }


}



void AckTable::deleteEntry(AckTable *rt, std::vector<AckOpt>::iterator r){
    std::this_thread:: sleep_for (std:: chrono:: milliseconds (MaintHoldoffTime));
    std::cout << "Deleting ack " << *r << " from table." << std::endl;
    rt->ackTableVector.erase(r);
}


std::ostream &operator<<(std::ostream &output, const AckTable & _table){

    output<<"Ack Timeout Table:"<<std::endl;

    for (std::vector<AckOpt>::const_iterator it = _table.ackTableVector.begin(); it !=
_table.ackTableVector.end(); ++it){

        std::cout<<*it;

    }//for


    return output;
}

bool AckTable::operator==(const AckTable &right) const{
    bool aux = false;
    if(this->ackTableVector == right.ackTableVector){
        aux = true;
    }
    return aux;
}
//
```

```
//  Bash.h
//  Route Request
//
//  Created by Pedro Loami on 23/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__Bash__
#define __Route_Request__Bash__

#include <stdio.h>
#include <stdlib.h>
#include "Ipv4.h"

class Bash{
    char * ifconfig;
    char * addroute;
public:
    void setIfconfig(int _subinterface, Ipv4 _newiP, Ipv4 _mask,Ipv4 _dstaddr);
    void setAddroute(Ipv4 _target,Ipv4 _gw, int subinterface);
};
#endif /* defined(__Route_Request__Bash__) */
//
//  Bash.cpp
//  Route Request
//
//  Created by Pedro Loami on 23/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "Bash.h"

void Bash::setIfconfig(int _subinterface, Ipv4 _newiP, Ipv4 _mask,Ipv4 _dstaddr){
    char buffer[50];
            sprintf(ifconfig,"ifconfig wlan0:%d %s netmask %s dstaddr
%s",_subinterface,_newiP.getIpStr(),_mask.getIpStr(),_dstaddr.getIpStr());
    system(buffer);

}
void Bash::setAddroute(Ipv4 _target,Ipv4 _gw, int subinterface){
    char buffer[50];
    sprintf(buffer , "route add -host %s gateway %s
en1:%d",_target.getIpStr(),_gw.getIpStr(),subinterface);
    system(buffer);
```

```
}//
//  Cache.h
//  Route Request
//
//  Created by Pedro Loami on 15/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__Cache__
#define __Route_Request__Cache__

#include "Route.h"
#include "Ipv4.h"
#include <vector>
#include <algorithm>     //o linux só roda com isso

class Cache {
    std::vector<Route> routes;

public:

    void appendCache(Route);

    Route giveRouteTo(Ipv4);    //critério menor hop

    bool alreadyExists(Route);   //verifica se a rota para um determinado add já existe

    bool intermedAlreadyExists(Route _route);

    bool existsRouteTo(Ipv4);

    bool propagateBrreply(Route);

    void printCache()const;
    //void setIntermedCachedRReply(std::vector<Ipv4>);
};

#endif /* defined(__Route_Request__Cache__) */
//
//  Cache.cpp
//  Route Request
//
//  Created by Pedro Loami on 15/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
```

```
//

#include "Cache.h"


void Cache::appendCache(Route _route){
    if( !alreadyExists(_route)){
            this->routes.push_back(_route);
    }

}

Route Cache::giveRouteTo(Ipv4 _dest){
    Route dest;
    int cont=100000;
    for (std::vector<Route>::iterator it = this->routes.begin(); it != this->routes.end(); ++it){

        if(_dest == it->getDest()){

            if(it->getIntermed().size()<cont){

                dest.setRoute(it->getDest(), it->getIntermed());

                cont=(int)it->getIntermed().size();

            }//if2

        }//if1

    }//for



    return dest;
}

bool Cache::alreadyExists(Route _route){
    std::vector<Route>::iterator it;

    it = find (this->routes.begin(),this->routes.end(),_route);
    if(it != routes.end()){
        return true;
    }
    else{
```

```
        return false;
    }
}

//esse método responde se o intermed da rota já existe no cache???

bool Cache::intermedAlreadyExists(Route _route){
    std::vector<Route>::iterator it;
    bool aux = false;
    Route dest;
    for (std::vector<Route>::iterator it = this->routes.begin(); it != this->routes.end(); ++it){

        if(_route.getIntermed() == it->getIntermed()){
            aux = true;

        }//if

    }//for


    return aux;
}

bool Cache::existsRouteTo(Ipv4 _dest){
    Route dest;
    bool answ = false;

    for (std::vector<Route>::iterator it = this->routes.begin(); it != this->routes.end(); ++it){

        if(_dest == it->getDest()){

            answ=true;

        }//if1

    }//for

    return answ;
}
/* This method compares two routes and verify if it can by concatenated to form
 * the Cahced RReply(return 0), or if it must be propagated RReq(return 1)
 */
bool Cache::propagateBrreply(Route routeRReq){
    bool ans;
```

```
    if(alreadyExists(routeRReq)){ //se conhece rota, verificar se existe elemento repetido
        Route rotaDoCache = giveRouteTo(routeRReq.getDest());
        std::vector<Ipv4> aux;

        aux.reserve( rotaDoCache.getIntermed().size() + routeRReq.getIntermed().size() ); //
preallocate memory
        aux.insert( aux.end(), rotaDoCache.getIntermed().begin(),
rotaDoCache.getIntermed().end() );
        aux.insert( aux.end(), routeRReq.getIntermed().begin(), routeRReq.getIntermed().end() );
        //XXXXXXXXXXXXXXX                    NAO SEI FAZER!!!
    }
    else{
        ans = true;//se o camarada nao conhece a rota entao propaga RReq

    }

    return ans;
}
void Cache::printCache()const{
    //Route dest;
    std::cout<<"~ ~ ~ ~ ~ ~ ~ ~Cache:~ ~ ~ ~ ~ ~ ~ ~\n"<<std::endl;
    std::cout<<"Cache:\n";
    int cont =0;

    for (std::vector<Route>::const_iterator it = this->routes.begin(); it != this->routes.end(); ++it){

        it->printRoute(++cont);
    }//for


    std::cout<<"~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ \n"<<std::endl;
    //return dest;
}//
//  Context.h
//  Route Request
//
//  Created by Pedro Loami on 15/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__Context__
#define __Route_Request__Context__
```

```
#include "Ipv4.h"
#include "DSRProcessor.h"
#include "Cache.h"
#include "Route.h"
#include "SocketRaw.h"

#include <sys/socket.h>
#include <chrono>
#include <thread>
#include <ctime>

#include "Interface.h"
#include "RReqTable.h"
#include "ackTable.h"

#include "RReply.h"
#include "RReq.h"
#include "RERROR.h"

#include "AckReqOpt.h"
#include "AckOpt.h"
#include "SourceRouteOpt.h"
#include "FixedHeaderDSR.h"

class Context{
    Message *objMessage;
    MessageMaintenance *objMessageMaintenance;

    RReq objRReq;
    RReply objRReply;

    RERROR objRError;
    AckOpt objAckOpt;
    AckReqOpt objAckReqOpt;
    SourceRouteOpt objSourceRouteOpt;

    Interface interface;

    Ipv4 mask;
    Ipv4 selfIp;
    Ipv4 bdcastAddr;

public:
    Context();
```

```
    //void processMessage(unsigned char *, Cache &_cache, Ipv4 _sender, RReqTable & );

    void processInput(unsigned char * buffer_in,Cache &_cache,Ipv4 _sender,RReqTable &
_rreqTable,AckTable &,u_int16_t &_gemRReq,u_int16_t &_gemSR);

    void processRReq(unsigned char * buffer_in,Cache &_cache,Ipv4 _sender,RReqTable &
_rreqTable, u_int16_t &_gemRReq, uint16_t & _gemSR);

    void processRReqRReply(int payLoadLen,unsigned char * buffer_in,Cache &_cache,Ipv4
_sender,RReqTable & _rreqTable, u_int16_t &_gemRReq, uint16_t & _gemSR);

    unsigned char * filterRouteRequest( unsigned char * filteredBUffer,int payLoadLen);

    void processReqAck(unsigned char * filteredBuffer, Ipv4 _sender);

    void processAck(unsigned char *filteredBuffer, AckTable &_tabelaAck);

    void processRRep(unsigned char *filteredBuffer,Ipv4 _sender, Cache &_cache);

    void processError(unsigned char *filteredBuffer, Ipv4 _sender, Cache &_cache);

    unsigned char * filterSourceRoute(unsigned char *filteredBuffer,int payLoadLen);

    bool confirmNode(Ipv4 _addr,uint16_t & _gemAckReq ,AckTable &_tabelaAck);

    void sleep(int _time){
        std:: this_thread:: sleep_for (std:: chrono:: milliseconds (_time));

    }
};

#endif /* defined(__Route_Request__Context__) */

//
//  Context.cpp
//  Route Request
//
//  Created by Pedro Loami on 15/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//


#include "Context.h"
```

```
int globalFlag = 0;

bool flagTimeout = false;

int flag = 0;


void pause_thread()
{
        std::this_thread::sleep_for (std::chrono::milliseconds(AckTimout));
        flagTimeout = true;
        std::cout<<"timeout"<<std::endl;
}


char * traduzVetor(unsigned char* buffer_in){
    int tam = buffer_in[2]>>8 | buffer_in[3];
    char * buffer_out = new char [tam+2];
    for(int j =0;j<tam+2;j++){
        buffer_out[j] = buffer_in[j];
    }
    return buffer_out;
}
Context::Context(){
    interface.setInterface();
    this->selfIp = DSRProcessor::getInstance().getSelfIp();
    this->mask = DSRProcessor::getInstance().getSelfMask();
    this->bdcastAddr = this->selfIp.makeBroadCastAddr(this->mask);
}

void Context::processRReqRReply(int payLoadLen, unsigned char * buffer_in,Cache
&_cache,Ipv4 _sender,RReqTable & _rreqTable, u_int16_t &_gemRReq, uint16_t &_gemSR){
   // Vai ser basicamente a mesma coisa do RReq tradicional mas conservando o RReply que
será processado apenas quando chegar ao destino
    unsigned char * bufferRReply = filterRouteRequest(buffer_in, payLoadLen);
    Interface interface_in;


    FixedHeaderDSR outputHeader;
    std::vector<Ipv4> emptyVector;
    Route routeToSender(_sender, emptyVector); //rota somente com o target setado ->para o
sender

    //        # verificar se a mensagem é expúria
```

```
    //        |       # verificar se eu sou o target
    //        |       |
    //        |       | -> sou ->   # verificar se eu tenho a rota para o sender
    //        |       |             | -> tenho -> Usar Sorce Route
    //        |       |             | -> não tenho -> Usar RReq+RReply
    //        |       |             #
    //        |       |
    //        |       | -> ~sou ->  # verificar se eu tenho a rota para o target
    //        |       |             | -> tenho ->  Cached RReply                  ->pode ser num route dicovery
ou piggybacked
    //        |       |             | -> não tenho -> Encaminhar RReq
    //        |       |             #
    //        |       |
    //        |       #
    //        |
    //        #
    //        break;
    //
    //
    this->objMessage = &this->objRReq;
    objMessage->lerBuffer(buffer_in);

    unsigned char * id = objMessage->getId();

    RReqTableType rreqIn(id[0],id[1],objMessage->getTarget());

    //std::cout<<"Teste das RReqs:\n"<<_rreqTable<<"\n##############\n"<<rreqIn<<std::endl;

    if(!_rreqTable.isThere(rreqIn)){ //Testa se é uma mensagem expúria
        std::cout<<"---------------\nReceived\n---------------\n"<<*objMessage;
        _rreqTable.appendRReq(rreqIn);

        //VERIFY IF THIS HOP IS THE TARGET:
        int flagTarget = 0;
        Ipv4 incomingAdd;
        for(int j=1;j<interface_in.getSizeInterface();j++){
            if(objMessage->getTarget() == interface_in.getIpInterfaceByOrder(j)){
                flagTarget =1;
                incomingAdd = interface_in.getIpInterfaceByOrder(j);
            }
        }
        if(flagTarget){//selfIp == objMessage->getTarget()){
            //std::cout<<"SouTarget!!"<<std::endl;
```

```
        Message *auxObj;
        auxObj = &objRReply;
        auxObj -> setIntermed(objMessage->getIntermed());
        auxObj -> addNodeIntermed(selfIp);
        auxObj -> setL(false);

        processRRep(bufferRReply, _sender, _cache);
        std::cout<<"------------------\nReceived(Piggybacked) from:"<<_sender<<"\n------------------\n"<<objRReply;


        if(_cache.alreadyExists(routeToSender)){//tenho a rota
            //RReply UNICAST - Via Source Route
            char * bufferRReply = auxObj->bufferSaida();

            objSourceRouteOpt.setSegsLeft(routeToSender.getIntermed().size());

            std::vector<Ipv4>auxIntermed;
            auxIntermed.push_back(incomingAdd);

            std::vector<Ipv4>concAux;
            concAux.reserve( auxIntermed.size() + routeToSender.getIntermed().size() ); // preallocate memory
            concAux.insert( concAux.end(), auxIntermed.begin(), auxIntermed.end() );
            concAux.insert( concAux.end(), routeToSender.getIntermed().begin(), routeToSender.getIntermed().end() );

            objSourceRouteOpt.setIntermed(concAux);


            outputHeader.appendBufferSaida(objSourceRouteOpt.bufferSaida(), 2+objSourceRouteOpt.getDataLen());
            outputHeader.appendBufferSaida(bufferRReply, auxObj->getDataLen()+2);

            char *bufferRReplyWHeader = outputHeader.bufferSaida();


            SocketRaw sock1(this->selfIp,8080,_sender,4950);
            sock1.enviar(bufferRReplyWHeader, (unsigned int) outputHeader.getNumericPayloadLength()+4);

            std::cout<<"....................................."<<std::endl;
            std::cout<<"Sent RReply with Source Route to:"<<_sender<<std::endl;
            std::cout<<objSourceRouteOpt<<std::endl;
```

```
            std::cout<<*auxObj<<std::endl;
            std::cout<<"....."<<std::endl;
        }
        else{//não tenho a rota...
            //RReply PIGGYbacked

            char * bufferRReplyP = auxObj->bufferSaida();//objMessage-
>bufferSaidaPiggy(auxObj->bufferSaida());

            for(int j = 1; j<interface_in.getSizeInterface();j++){
                RReq auxRReqObj;

                auxRReqObj.setId(_gemRReq);
                _gemRReq+=1;

                auxRReqObj.setL(false);
                auxRReqObj.setTarget(_sender);
                auxRReqObj.setIntermed(emptyVector);
                auxRReqObj.addNodeIntermed(interface.getIpInterface()[j]);

                outputHeader.appendBufferSaida(auxRReqObj.bufferSaida(),
auxRReqObj.getDataLen()+2);
                outputHeader.appendBufferSaida(auxObj->bufferSaida(), auxObj-
>getDataLen()+2);

                char * bufferRReplyPiggy = outputHeader.bufferSaida();
                SocketRaw sock2(objMessage-
>getTarget(),8080,interface.getIpInterfaceByOrder(j).makeBroadCastAddr(interface.getNetmask
()),4950);
                sock2.enviar(bufferRReplyPiggy, outputHeader.getNumericPayloadLength()+4);
                outputHeader.clearAppendedOpts();

                std::cout<<"....."<<std::endl;
                std::cout<<"Sent piggybacked RReply at:
"<<interface.getIpInterfaceByOrder(j).makeBroadCastAddr(interface.getNetmask())<<std::endl;
                std::cout<<"With RReq:\n"<<auxRReqObj<<std::endl;
                std::cout<<"....."<<std::endl;

            }
            std::cout<<*auxObj<<std::endl;
            std::cout<<"....."<<std::endl;
        }//else do verifica cached route
```

```
        }//if verify target
        else{//Não sou target...
            if(_cache.existsRouteTo(objMessage->getTarget())){//tenho a rota?
                //Cached RReply
                objSourceRouteOpt.setSegsLeft(routeToSender.getIntermed().size());
                objSourceRouteOpt.setIntermed(routeToSender.getIntermed());

                outputHeader.appendBufferSaida(objSourceRouteOpt.bufferSaida(),
2+objSourceRouteOpt.getDataLen());

                Route auxRoute = _cache.giveRouteTo(objMessage->getTarget());

                Message *auxObj;
                auxObj = &objRReply;


                auxObj->setIntermed(auxRoute.appendInter(objMessage->getIntermed()));
                auxObj->addNodeIntermed(selfIp);
                auxObj->addNodeIntermed(objMessage->getTarget());
                auxObj->setL(false);

                char * bufferCachedRReply = auxObj->bufferSaida();

                outputHeader.appendBufferSaida(bufferCachedRReply, auxObj->getDataLen()+2);
                char *bufferCachedRReplyHeader = outputHeader.bufferSaida();

                SocketRaw sock3(this->selfIp,8080,_sender,4950);
                sock3.enviar(bufferCachedRReplyHeader,
4+outputHeader.getNumericPayloadLength());

                std::cout<<"....................................."<<std::endl;
                std::cout<<"Cached RReply with Source Route sent to:"<<_sender<<std::endl;
                std::cout<<objSourceRouteOpt<<std::endl;
                std::cout<<*auxObj<<std::endl;
                std::cout<<"....................................."<<std::endl;
            }
            else{//não tenho a rota..
                //RReq retransmition
                this->objMessage->addNodeIntermed(this->selfIp);

                char * bufferRetransmission = objMessage->bufferSaida();
                outputHeader.appendBufferSaida(bufferRetransmission, 2+objMessage-
>getDataLen());
```

```
            char * bufferRetransmissionHeader = outputHeader.bufferSaida();
            for(int j = 1; j<interface.getSizeInterface();j++){


                SocketRaw
sock4(_sender,8080,interface.getIpInterfaceByOrder(j).makeBroadCastAddr(interface.getNetma
sk()),4950);
                sock4.enviar(bufferRetransmissionHeader,
4+outputHeader.getNumericPayloadLength());

                std::cout<<"..............................."<<std::endl;
                std::cout<<"RReq target to:"<<objMessage->getTarget()<<"from:"<<_sender<<"
retransmitted
at:"<<interface.getIpInterfaceByOrder(j).makeBroadCastAddr(interface.getNetmask())<<std::end
l;
                std::cout<<"......."<<std::endl;
            }//for
            std::cout<<*objMessage<<std::endl;
            std::cout<<"..............................."<<std::endl;
        }//else - tenho a rota?
    }//else verify target




  }//fim if rreqTable
  else{
    std::cout<<"Mensagem expuria descartada."<<std::endl;
  }

}

void Context::processRReq(unsigned char* filteredBuffer,Cache &_cache,Ipv4
_sender,RReqTable & _rreqTable, u_int16_t &_gem,uint16_t & _gemSR){
  FixedHeaderDSR outputHeader;
  std::vector<Ipv4> emptyVector;
  Route routeToSender(_sender, emptyVector); //rota somente
  this->objMessage = &this->objRReq;
  objMessage->lerBuffer(filteredBuffer);

  unsigned char * id = objMessage->getId();
```

```
    RReqTableType rreqIn(id[0],id[1],objMessage->getTarget());

    //std::cout<<"Teste das RReqs:\n"<<_rreqTable<<"\n##############\n"<<rreqIn<<std::endl;

    if(!_rreqTable.isThere(rreqIn)){ //Testa se é uma mensagem expúria
        std::cout<<"---------------\nReceived\n---------------\n"<<*objMessage;
        _rreqTable.appendRReq(rreqIn);

        //VERIFY IF THIS HOP IS THE TARGET:
        int flagTarget = 0;
        Ipv4 incomingAdd;
        for(int j=1;j<interface.getSizeInterface();j++){
            if(objMessage->getTarget() == interface.getIpInterfaceByOrder(j)){
                flagTarget =1;
                incomingAdd = interface.getIpInterfaceByOrder(j);
            }
        }
        if(flagTarget){//selfIp == objMessage->getTarget()){
            //std::cout<<"SouTarget!!"<<std::endl;

            Message *auxObj;
            auxObj = &objRReply;
            auxObj -> setIntermed(objMessage->getIntermed());
            auxObj -> addNodeIntermed(selfIp);
            auxObj -> setL(false);

            if(_cache.alreadyExists(routeToSender)){//tenho a rota
                //RReply UNICAST - Via Source Route
                char * bufferRReply = auxObj->bufferSaida();

                objSourceRouteOpt.setSegsLeft(routeToSender.getIntermed().size());

                std::vector<Ipv4>auxIntermed;
                auxIntermed.push_back(incomingAdd);

                std::vector<Ipv4>concAux;
                concAux.reserve( auxIntermed.size() + routeToSender.getIntermed().size() ); //
preallocate memory
                concAux.insert( concAux.end(), auxIntermed.begin(), auxIntermed.end() );
                concAux.insert( concAux.end(), routeToSender.getIntermed().begin(),
routeToSender.getIntermed().end() );

                objSourceRouteOpt.setIntermed(concAux);
```

```
            outputHeader.appendBufferSaida(objSourceRouteOpt.bufferSaida(),
2+objSourceRouteOpt.getDataLen());
            outputHeader.appendBufferSaida(bufferRReply, auxObj->getDataLen()+2);

            char *bufferRReplyWHeader = outputHeader.bufferSaida();


            SocketRaw sock1(this->selfIp,8080,_sender,4950);
            sock1.enviar(bufferRReplyWHeader, (unsigned int)
outputHeader.getNumericPayloadLength()+4);

            std::cout<<"..............................."<<std::endl;
            std::cout<<"Sent RReply with Source Route to:"<<_sender<<std::endl;
            std::cout<<objSourceRouteOpt<<std::endl;
            std::cout<<*auxObj<<std::endl;
            std::cout<<"..............................."<<std::endl;
        }
        else{//não tenho a rota...
            //RReply PIGGYbacked

            //char * bufferRReplyP = auxObj->bufferSaida();//objMessage-
>bufferSaidaPiggy(auxObj->bufferSaida());
            RReqTableType rreqOut;
            for(int j = 1; j<interface.getSizeInterface();j++){
                RReq auxRReqObj;

                auxRReqObj.setId(_gem);
                _gem+=1;

                auxRReqObj.setL(false);
                auxRReqObj.setTarget(_sender);
                auxRReqObj.setIntermed(emptyVector);
                auxRReqObj.addNodeIntermed(interface.getIpInterface()[j]);

                outputHeader.appendBufferSaida(auxRReqObj.bufferSaida(),
auxRReqObj.getDataLen()+2);
                outputHeader.appendBufferSaida(auxObj->bufferSaida(), auxObj-
>getDataLen()+2);

                char * bufferRReplyPiggy = outputHeader.bufferSaida();
                SocketRaw sock2(objMessage-
>getTarget(),8080,interface.getIpInterfaceByOrder(j).makeBroadCastAddr(interface.getNetmask
()),4950);
```

```
                sock2.enviar(bufferRReplyPiggy, outputHeader.getNumericPayloadLength()+4);
                outputHeader.clearAppendedOpts();

                std::cout<<"..............................."<<std::endl;
                std::cout<<"Sent piggybacked RReply at:
"<<interface.getIpInterfaceByOrder(j).makeBroadCastAddr(interface.getNetmask())<<std::endl;
                std::cout<<"With RReq:\n"<<auxRReqObj<<std::endl;
                std::cout<<"....."<<std::endl;

                unsigned char * auxId = auxRReqObj.getId();
                //RReqTableType rreqOut(auxId[0],auxId[1],auxRReqObj.getTarget());
                rreqOut.setIdNumber(auxId[0], auxId[1]);
                rreqOut.setTarget(_sender);
                _rreqTable.appendRReq(rreqOut);
            }
            std::cout<<*auxObj<<std::endl;
            std::cout<<"..............................."<<std::endl;
        }//else do verifica cached route



    }//if verify target
    else{//Não sou target...
        if(_cache.existsRouteTo(objMessage->getTarget())){//tenho a rota?
            //Cached RReply
            objSourceRouteOpt.setSegsLeft(routeToSender.getIntermed().size());
            objSourceRouteOpt.setIntermed(routeToSender.getIntermed());

            outputHeader.appendBufferSaida(objSourceRouteOpt.bufferSaida(),
2+objSourceRouteOpt.getDataLen());

            Route auxRoute = _cache.giveRouteTo(objMessage->getTarget());

            Message *auxObj;
            auxObj = &objRReply;


            auxObj->setIntermed(auxRoute.appendInter(objMessage->getIntermed()));
            auxObj->addNodeIntermed(selfIp);
            auxObj->addNodeIntermed(objMessage->getTarget());
            auxObj->setL(false);

            char * bufferCachedRReply = auxObj->bufferSaida();
```

```
            outputHeader.appendBufferSaida(bufferCachedRReply, auxObj->getDataLen()+2);
            char *bufferCachedRReplyHeader = outputHeader.bufferSaida();

            SocketRaw sock3(this->selfIp,8080,_sender,4950);
            sock3.enviar(bufferCachedRReplyHeader,
4+outputHeader.getNumericPayloadLength());

            std::cout<<"....................................."<<std::endl;
            std::cout<<"Cached RReply with Source Route sent to:"<<_sender<<std::endl;
            std::cout<<objSourceRouteOpt<<std::endl;
            std::cout<<*auxObj<<std::endl;
            std::cout<<"....................................."<<std::endl;
        }
        else{//não tenho a rota..
            //RReq retransmition
            this->objMessage->addNodeIntermed(this->selfIp);

            char * bufferRetransmission = objMessage->bufferSaida();
            outputHeader.appendBufferSaida(bufferRetransmission, 2+objMessage-
>getDataLen());

            char * bufferRetransmissionHeader = outputHeader.bufferSaida();
            for(int j = 1; j<interface.getSizeInterface();j++){


                SocketRaw
sock4(_sender,8080,interface.getIpInterfaceByOrder(j).makeBroadCastAddr(interface.getNetma
sk()),4950);
                sock4.enviar(bufferRetransmissionHeader,
4+outputHeader.getNumericPayloadLength());

                std::cout<<"....................................."<<std::endl;
                std::cout<<"RReq target to:"<<objMessage->getTarget()<<"from:"<<_sender<<"
retransmitted
at:"<<interface.getIpInterfaceByOrder(j).makeBroadCastAddr(interface.getNetmask())<<std::end
l;
                std::cout<<"......."<<std::endl;
            }//for
            std::cout<<*objMessage<<std::endl;
            std::cout<<"....................................."<<std::endl;
        }//else - tenho a rota?
    }//else verify target

  }//fim if rreqTable
```

```
    else{
        std::cout<<"Mensagem expuria descartada."<<std::endl;
    }

}

unsigned char * Context::filterSourceRoute(unsigned char * filteredBUffer,int payLoadLen){
    this->objSourceRouteOpt.lerBuffer(filteredBUffer);
    int tam = this->objSourceRouteOpt.getDataLen();

    unsigned char * buffer2 = new unsigned char [payLoadLen - tam-2];

    for(int j = 0;j<payLoadLen-tam-2;j++){
        buffer2[j] = filteredBUffer[j+tam+2];
    }

    return buffer2;

}

unsigned char * Context::filterRouteRequest(unsigned char * filteredBUffer,int payLoadLen){
    //lê o rreq retorna o route reply piggybacked num rreq

    this->objRReq.lerBuffer(filteredBUffer);
    int tam = this->objRReq.getDataLen();

    unsigned char * buffer2 = new unsigned char [payLoadLen - tam-2];

    for(int j = 0;j<payLoadLen-tam-2;j++){
        buffer2[j] = filteredBUffer[j+tam+2];
    }

    return buffer2;

}

void Context::processReqAck(unsigned char * filteredBuffer, Ipv4 _sender){

    FixedHeaderDSR outputHeader;

    this->objAckReqOpt.lerBuffer(filteredBuffer);
    std::cout<<"----------------------\nProcessing ReqAck from:"<<_sender<<"\n----------------------
\n"<<objAckReqOpt<<"--------------"<<std::endl;
```

```
    //Setup Answer
    uint16_t idt16 = this->objAckReqOpt.getNumericIdt();
    this->objAckOpt.setIdt(*&idt16);
    this->objAckOpt.setAckDestAdd(_sender);
    this->objAckOpt.setAckSourceAdd(this->selfIp);
    std::cout<<objAckOpt<<std::endl;

    char * bufferAck = this->objAckOpt.bufferSaida();
    outputHeader.appendBufferSaida(bufferAck, 2+objAckOpt.getDataLen());

    //Send Unicast
    SocketRaw sock160(this->selfIp,8080,_sender,4950);
    sock160.enviar(outputHeader.bufferSaida(), (unsigned
int)outputHeader.getNumericPayloadLength()+4);

    std::cout<<"Sent Ack unicast to:"<<_sender<<std::endl;
    std::cout<<objAckOpt<<std::endl;

}

void Context::processAck(unsigned char * filteredBuffer, AckTable & _tabelaAck ){
    //Recebe um ack e coloca no banco
    this->objAckOpt.lerBuffer(filteredBuffer);
    _tabelaAck.appendAckReq(this->objAckOpt);

}

bool Context::confirmNode(Ipv4 _addr,uint16_t & _gemAckReq,AckTable & _tabelaAck){
    objAckReqOpt.setIdt(_gemAckReq);
    //_gemAckReq++;

    FixedHeaderDSR outputHeader;

    char * buffer = new char[objAckReqOpt.getDataLen()+4];
    buffer = objAckReqOpt.bufferSaida();

    char * bufferOut = new char[objAckReqOpt.getDataLen()+6];
    outputHeader.appendBufferSaida(buffer,objAckReqOpt.getDataLen()+2);
    bufferOut = outputHeader.bufferSaida();

    for(int j =1;j<interface.getSizeInterface();j++){
        unsigned char  auxVecInt = interface.getIpInterfaceByOrder(j).getIpByNumber(0);
        if(auxVecInt==192 || auxVecInt == 10){
            SocketRaw sockAckOut(interface.getIpInterfaceByOrder(j),8080,_addr,4950);
```

```
            sockAckOut.enviar(bufferOut, outputHeader.getNumericPayloadLength()+4);

            std::cout<<"....................................."<<std::endl;
            std::cout<<"Sent AckReq from: "<<interface.getIpInterfaceByOrder(j)<<" to:
"<<_addr<<std::endl;
            std::cout<<objAckReqOpt<<std::endl;
            std::cout<<"....................................."<<std::endl;
        }
    }
    std::thread t1 (pause_thread);

    AckOpt auxObj;
    auxObj.setAckSourceAdd(_addr);
    auxObj.setAckDestAdd(interface.getIpInterfaceByOrder(1));
    auxObj.setIdt(_gemAckReq);
    _gemAckReq++;

    bool flagLocal = false;

    while(!flagTimeout){                    //while(_tabelaAck.isThere(auxObj)){
        // de repente poe um temporizador pequeno aqui.
        if(_tabelaAck.isThere(auxObj)){
            //recebi o ack
            flagLocal=1;
            //return true;
            //break;
        }
        else{
            //continua aguardando
        }
    }
    if(flagLocal == 0 ){
        //ERRO -> enviar route error
//        objRError.setErrorType(1);
//        objRError.setErrorSourceAdd(interface.getIpInterfaceByOrder(1));
//        objRError.setErrorDestAdd(_sender);
//        objRError.setUnreachableDestAdd(_addr);
        std::cout<<"Envia RRerror"<<std::endl;
        //return false;
        //break;
    }
    else{
        //return true;
    }
```

```
    t1.join();

    return flagLocal;
}

void Context::processError(unsigned char* bufferFiltered, Ipv4 _sender, Cache &_cache){
    //TODO: RETIRAR UM ERRO E ELIMINAR A ROTA CORRESPONDENTE.
EVENTUALMENTE CHAMAR UM ROUTE DISCOVERY.
}

void Context::processRRep(unsigned char* filteredBuffer,Ipv4 _sender, Cache & _cache){
    objMessage = &objRReply;
    objMessage->lerBuffer(filteredBuffer);
    std::cout<<"----------------------\nProcessing RReply from: "<<_sender<<"\n----------------------
\n"<<*objMessage<<"--------"<<std::endl;

    std::vector<Ipv4>::iterator it = objMessage->getIntermed().end();
    --it;

    Route rota(*it,objMessage->getIntermed());
    _cache.appendCache(rota);
    //std::cout<<"linux> route add -host "<<rota.getDest()<<" gw "<<rota.getIntermed().at(0)<<"
wlan0:"<<2<<std::endl;
    int auxGamb = (int)rota.getIntermed().size();

    //std::cout<<objMessage->getIntermed().at(1)<<" -:- "<<objMessage-
>getIntermed().at(0)<<std::endl;
    interface.addRoute(objMessage->getIntermed().at(1), objMessage->getIntermed().at(0));
    //_cache.printCache();

}

void Context::processInput(unsigned char* buffer_in, Cache &_cache,Ipv4 _sender,RReqTable
& _rreqTable,AckTable & _tabelaAck, u_int16_t &_gemRReq,uint16_t & _gemSR){

    //Interface interface_Local;
    //interface_Local.setInterface();
    FixedHeaderDSR inputHeader;//,outputHeader;

    int parseResult = inputHeader.parseHeader(buffer_in);

    unsigned char * filteredBuffer = inputHeader.bufferFilter(buffer_in);

//    std::vector<Ipv4> emptyVector;
```

```
//    Route routeToSender(_sender, emptyVector); //rota somente com o target setado ->para o
sender


    switch(parseResult){
        case 1:{    //RReq
            this->processRReq(filteredBuffer, _cache, _sender, _rreqTable, _gemRReq,_gemSR);
            break;
        }
        case 3:{    //RReq +RReply
            this->processRReqRReply(inputHeader.getNumericPayloadLength(), filteredBuffer,
_cache, _sender, _rreqTable, _gemRReq, _gemSR);

            break;
        }
        case 32:{    //Ack
            std::cout<<"Entrou"<<std::endl;
            this->processAck(filteredBuffer, _tabelaAck);

            break;
        }
        case 96:{    //Ack
//            std::cout<<"Chegou Dados"<<std::endl;
//            objSourceRouteOpt.lerBuffer(filteredBuffer);
//            int n= objSourceRouteOpt.getSegsLeft();
//            Ipv4 prox = objSourceRouteOpt.getIntermed().at(n-1);
//            objSourceRouteOpt.setSegsLeft(objSourceRouteOpt.getSegsLeft()-1);
//
//            SocketRaw sock98(inputAdd,8080, _sender,4950);
//            sock98.enviar(outputHeader.bufferSaida(), (unsigned
int)outputHeader.getNumericPayloadLength()+4);
//            //TODO
//            std::cout<<"--------------------"<<"Sent Erro unicast to:"<<_sender<<std::endl;
//            std::cout<<objRError<<std::endl;
//            std::cout<<"--------------------"<<std::endl;
//

            break;
        }
        case 98:{    //SRoute + RReply
            unsigned char * bufferRReply = this->filterSourceRoute(filteredBuffer,
inputHeader.getNumericPayloadLength());
            Ipv4 inputAdd;
            Interface interface_in;
```

```
            //this->objRReply.lerBuffer(bufferRReply);
            for(int j=0;j<interface_in.getSizeInterface();j++){
                int n = (int)(this->objSourceRouteOpt.getDataLen()-2)/4;
                Ipv4 auxIpv4 = this->objSourceRouteOpt.getIntermed().at(n-1);
                //Ipv4 auxIp4 = auxIpv4.at(n-1);
                //std::cout<<this->interface;
                Ipv4 ipInterface = interface_in.getIpInterfaceByOrder(j);
                if(ipInterface == auxIpv4){
                        //A mensagem se destina para mim, logo eu posso lê-la
                    inputAdd = interface_in.getIpInterfaceByOrder(j);
                    this->processRRep(bufferRReply, _sender, _cache);
                  }
                  else{
                    //A mensagem não é para mim, logo retransmito para o próximo camarada
                    std::cout<<"size: "<<objSourceRouteOpt.getIntermed().size()<<"Segs: "<<
(unsigned int)objSourceRouteOpt.getSegsLeft()<<" \nsub:
"<<objSourceRouteOpt.getIntermed().size() - (unsigned
int)objSourceRouteOpt.getSegsLeft()<<std::endl;
                    Ipv4 nextHop = this-
>objSourceRouteOpt.getIntermed().at(objSourceRouteOpt.getIntermed().size() - (unsigned
int)objSourceRouteOpt.getSegsLeft()-1);     //ERRO AQUI!
                    //confirmar next hop -> encaminhar ou devolver um erro
                    bool confirmacao = this->confirmNode(nextHop, _gemRReq, _tabelaAck);

                    if(confirmacao){
                          //transmite
                          FixedHeaderDSR outputHeader;
                          outputHeader.appendBufferSaida(traduzVetor(buffer_in), buffer_in[0]>>8 |
buffer_in[1] );
                    }
                    else{
                       //erro
                       FixedHeaderDSR outputHeader;

                       objRError.setErrorType(1);
                       objRError.setErrorSourceAdd(inputAdd);
                       objRError.setErrorDestAdd(_sender);
                       objRError.setUnreachableDestAdd(nextHop);


outputHeader.appendBufferSaida(objRError.bufferSaida(),objRError.getDataLen()+2);

                       //Send Unicast
                       SocketRaw sock98(inputAdd,8080, _sender,4950);
```

```
                    sock98.enviar(outputHeader.bufferSaida(), (unsigned
int)outputHeader.getNumericPayloadLength()+4);
                    //TODO
                    std::cout<<"--------------------"<<"Sent Erro unicast to:"<<_sender<<std::endl;
                    std::cout<<objRError<<std::endl;
                    std::cout<<"--------------------"<<std::endl;

                }

            }
          }
          break;
      }
      case 99:{    //SRoute + RError

          //unsigned char * bufferRError = this->filterSourceRoute(filteredBuffer,
inputHeader.getNumericPayloadLength());
          Ipv4 inputAdd;

          for(int j=1;j<interface.getSizeInterface();j++){

              if(interface.getIpInterfaceByOrder(j) == *this-
>objSourceRouteOpt.getIntermed().end()){
                  //A mensagem se destina para mim, logo eu posso lê-la
                  inputAdd = interface.getIpInterfaceByOrder(j);
                  //this->processErro();
              }
              else{
                  //A mensagem não é para mim, logo retransmito para o próximo camarada
                  std::cout<<"size: "<<objSourceRouteOpt.getIntermed().size()<<"Segs: "<<
(unsigned int)objSourceRouteOpt.getSegsLeft()<<" \nsub:
"<<objSourceRouteOpt.getIntermed().size() - (unsigned
int)objSourceRouteOpt.getSegsLeft()<<std::endl;
                  Ipv4 nextHop = this-
>objSourceRouteOpt.getIntermed().at(objSourceRouteOpt.getIntermed().size() - (unsigned
int)objSourceRouteOpt.getSegsLeft()-1);     //ERRO AQUI!
                  //confirmar next hop -> encaminhar ou devolver um erro
                  bool confirmacao = this->confirmNode(nextHop, _gemRReq, _tabelaAck);

                  if(confirmacao){
                      //transmite
                      FixedHeaderDSR outputHeader;
                      outputHeader.appendBufferSaida(traduzVetor(buffer_in), buffer_in[0]>>8 |
buffer_in[1] );
```

```
                }
                else{
                    //erro
                    FixedHeaderDSR outputHeader;

                    objRError.setErrorType(1);
                    objRError.setErrorSourceAdd(inputAdd);
                    objRError.setErrorDestAdd(_sender);
                    objRError.setUnreachableDestAdd(nextHop);


outputHeader.appendBufferSaida(objRError.bufferSaida(),objRError.getDataLen()+2);

                    //Send Unicast
                    SocketRaw sock98(inputAdd,8080, _sender,4950);
                    sock98.enviar(outputHeader.bufferSaida(), (unsigned
int)outputHeader.getNumericPayloadLength()+4);
                    //TODO
                    std::cout<<"--------------------"<<"Sent Erro unicast to:"<<_sender<<std::endl;
                    std::cout<<objRError<<std::endl;
                    std::cout<<"--------------------"<<std::endl;

                }

            }
        }
        break;
    }

    case 160:{    //ReqAck
        //When this node receive an ReqAck
        std::cout<<"Req Ack"<<std::endl;
        this->processReqAck(filteredBuffer, _sender);
        break;
    }


    default:{}
  }//switch
}

//
//  FixedHeaderDSR.h
//  Route Request
```

```
//
//  Created by Pedro Loami on 18/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__FixedHeaderDSR__
#define __Route_Request__FixedHeaderDSR__

#include "MessageMaintenance.h"
#include "Message.h"

class FixedHeaderDSR{
    uint8_t nextHeader;
    uint8_t flowStateAndReserved;
    uint8_t payloadLength[2];

    char *buffer_total;

public:
    FixedHeaderDSR();

    char *bufferSaida();
    void lerBuffer(unsigned char *buffer);

    unsigned char * bufferFilter(unsigned char *buffer_in);
    void appendBufferSaida(char * buffer_in, uint8_t size_in);
    void clearAppendedOpts();

    int parseHeader(unsigned char *);

    void setNextHeader(uint8_t);
    uint8_t getNextHeader();

    void setFlowStateAndReserved(bool);
    uint8_t getFlowStateAndReserved()const;

    void setPayloadLength(uint8_t,uint8_t);
    void setPayloadLength(uint16_t);
    uint8_t * getPayloadLength()const;
    uint16_t getNumericPayloadLength()const;


};
#endif /* defined(__Route_Request__FixedHeaderDSR__) */
```

```
//
//  FixedHeaderDSR.cpp
//  Route Request
//
//  Created by Pedro Loami on 18/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "FixedHeaderDSR.h"
FixedHeaderDSR::FixedHeaderDSR(){

    this->setNextHeader(17);
    this->setFlowStateAndReserved(false);
    this->setPayloadLength(0);

    buffer_total = new char [4*sizeof(uint8_t)];

    buffer_total[0] = 0x11;

    buffer_total[1] = 0x00;

    buffer_total[2] = 0x00;

    buffer_total[3] = 0x00;

}
char * FixedHeaderDSR::bufferSaida(){
//    char * buffer;
//
//    buffer= new char [4*sizeof(uint8_t)];
//
    buffer_total[0] = this->getNextHeader();

    buffer_total[1] = this->getFlowStateAndReserved();

    buffer_total[2] = this->getPayloadLength()[0];

    buffer_total[3] = this->getPayloadLength()[1];


    return buffer_total;

}
```

```
void FixedHeaderDSR::lerBuffer(unsigned char *buffer){
    this->setNextHeader(buffer[0]);
    this->setFlowStateAndReserved(buffer[1] & 0x80);
    this->setPayloadLength((uint8_t)buffer[2],(uint8_t)buffer[3]);

}

unsigned char * FixedHeaderDSR::bufferFilter(unsigned char *buffer_in){
    this->setPayloadLength((uint8_t)buffer_in[2],(uint8_t)buffer_in[3]);

//    std::cout<<"Teste bufferFilter - buffer_in:\nSize:"<<getNumericPayloadLength()<<std::endl;
//    for(int j=0;j<4+this->getNumericPayloadLength();j++){
//        std::cout<<""<<(unsigned int)buffer_in[j]<<std::endl;
//    }
//
    unsigned char * buffer;
    int tam = this->getNumericPayloadLength();


    buffer = new unsigned char [tam];

    for(int j=0;j<tam;j++){
        //view =buffer_in[j+4];
        //std::cout<<"teste:"<<(unsigned int)buffer_in[j+4]<<std::endl;
        buffer[j] = buffer_in[j+4];
    }
    return buffer;
}

void FixedHeaderDSR::appendBufferSaida(char * buffer_in, uint8_t size_in){
    int tam = 4+this->getNumericPayloadLength();
    setPayloadLength(getNumericPayloadLength()+size_in);
    char *bufferAux = new char[size_in + tam];

    for(int j=0;j<size_in + tam;j++){
        if(j<tam){
            bufferAux[j] = buffer_total[j];
        }
        else
            bufferAux[j] = buffer_in[j-tam];
    }

    buffer_total = bufferAux;
```

```
}

int FixedHeaderDSR::parseHeader(unsigned char * buffer_in){
    this->setPayloadLength((uint8_t)buffer_in[2],(uint8_t)buffer_in[3]);
    uint16_t totalLen = 4+this->getNumericPayloadLength();
    int parseSum = 0;
    int atualLength=0;
    int j = 4;

    while(j<totalLen){

        parseSum += buffer_in[j];
        atualLength = buffer_in[j+1];
        j+=atualLength +2;
    }

    return parseSum;
}

void FixedHeaderDSR::clearAppendedOpts(){
    buffer_total = new char[4];

    buffer_total[0] = this->getNextHeader();

    buffer_total[1] = this->getFlowStateAndReserved();

    buffer_total[2] = 0;

    buffer_total[3] = 0;

    this->setPayloadLength(0);
}
void FixedHeaderDSR::setNextHeader(uint8_t _in){
    this->nextHeader = _in;
}

uint8_t FixedHeaderDSR::getNextHeader() {
    //Como esse header vem logo depois do cabeçalho IP, o próximo header vai ser referente ao
UDP(17) e vai ser o default por enquanto.
    return this->nextHeader;
}

void FixedHeaderDSR::setFlowStateAndReserved(bool F){
    if(F == true){
```

```
        this->flowStateAndReserved = 0x80;
    }
    else
        this->flowStateAndReserved = 0x00;
}
uint8_t FixedHeaderDSR::getFlowStateAndReserved() const{
    return this->flowStateAndReserved;
}

void FixedHeaderDSR::setPayloadLength(uint16_t _gem){
    this->payloadLength[1] = _gem & 0xFF;  //menos significativo
    this->payloadLength[0] = _gem >> 8;    //mais significativo
}

void FixedHeaderDSR::setPayloadLength(uint8_t _a, uint8_t _b){
    this->payloadLength[1] = _b;
    this->payloadLength[0] = _a;

}
uint8_t * FixedHeaderDSR::getPayloadLength()const{
    static uint8_t * aux;
    aux = new uint8_t[2]();
    aux[0] = this->payloadLength[0];
    aux[1] = this->payloadLength[1];

    return aux;
}
uint16_t FixedHeaderDSR::getNumericPayloadLength()const{
    return this->payloadLength[0]<<8 | this->payloadLength[1];
};//
//  HopDevice.h
//  Cliente UDP
//
//  Created by Pedro Loami on 11/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Cliente_UDP__HopDevice__
#define __Cliente_UDP__HopDevice__


#include <unistd.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
```

```
#include <net/if.h>
//------------------------
#include <stdio.h>
#include <sys/types.h>
#include <ifaddrs.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>

#include "Ipv4.h"
#include "string"

#include <iostream>
#include <vector>

class HopDevice{
    Ipv4 hopAddr;
    Ipv4 mask;
    Ipv4 bdcastAddr;
    char *interface;

    void engine();

public:
    HopDevice();
    HopDevice(char * _interface);

    void initComm();
    void refreshInfo();
    Ipv4 getIp();
    Ipv4 getmask();

    char * getInterface();
    void setInterface(char *);

    Ipv4 get_bdcastAddr();
    void set_bdcastAddr();

    std::vector<Ipv4> getCompleteInterface(bool _ipv6);

};

#endif /* defined(__Cliente_UDP__HopDevice__) */
//
```

```
//  HopDevice.cpp
//  Cliente UDP
//
//  Created by Pedro Loami on 11/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "HopDevice.h"


HopDevice::HopDevice(char * _interface){
    this->interface = new char[10]();
    this->setInterface(_interface);
    engine();
}

void HopDevice::initComm(){
    Ipv4 ip, netmask;

    char * answ = (char *)malloc(sizeof(char)*10);

    std::cout << "Inicializacao da maquina:" << std::endl;
    std::cout << "Entre com o nome da interface ativa:" << std::endl;
    std::cin  >> answ;

    setInterface( answ );

    std::cout << "Interface escolhida:" << this->interface << std::endl;

    engine();

    ip = this->getIp();
    netmask = this->getmask();

    std::cout << "Rede ativa com:" <<std::endl;
    std::cout << "IP:" << ip << std::endl;
    std::cout << "netmask:" << netmask << std::endl;
}

void HopDevice::engine(){

    int fd, fd_n;
    struct ifreq ifr;
    struct ifreq ifr_nmask;
```

```
    fd = socket(AF_INET, SOCK_DGRAM, 0);
    fd_n = fd;

    /* I want to get an IPv4 IP address */
    ifr.ifr_addr.sa_family = AF_INET;
    ifr_nmask.ifr_addr.sa_family = AF_INET;

    /* I want IP address attached to "eth0" */
    strncpy(ifr.ifr_name, interface, IFNAMSIZ-1);
    strncpy(ifr_nmask.ifr_name, interface, IFNAMSIZ-1);

    if(ioctl(fd, SIOCGIFADDR, &ifr) <0 || ioctl(fd_n, SIOCGIFNETMASK, &ifr_nmask) <0){
        printf("erro:interface %s inexistente ou offline",interface);
        exit(0);
    }
    close(fd);
    close(fd_n);

    hopAddr.setIpCharPtr(inet_ntoa(((struct sockaddr_in *)&ifr.ifr_addr)->sin_addr));
    mask.setIpCharPtr(inet_ntoa(((struct sockaddr_in *)&ifr_nmask.ifr_addr)->sin_addr));
    bdcastAddr=hopAddr.makeBroadCastAddr(mask);
}

HopDevice::HopDevice(){
}

Ipv4 HopDevice::get_bdcastAddr(){
    return this->bdcastAddr;
}

void HopDevice::set_bdcastAddr(){
    bdcastAddr = hopAddr.makeBroadCastAddr(mask);
}

 void HopDevice::refreshInfo(){
    engine();
}

Ipv4 HopDevice::getIp(){
    return hopAddr;
}

Ipv4 HopDevice::getmask(){
```

```
    return mask;
}

char * HopDevice::getInterface(){
    return interface;
}

void HopDevice::setInterface(char * _interface){
    if(!this->interface) this->interface = new char[10]();

    strcpy(interface, _interface);
}


//método deveria ter compatibilidade com ipv6, mas como eu não criei a clase ipv6, fica para a próxima, logo por default, entre com _ipv6 = false
std::vector<Ipv4> HopDevice::getCompleteInterface(bool _ipv6){

        std::vector<Ipv4> interfaceC;


        struct ifaddrs * ifAddrStruct=NULL;
        struct ifaddrs * ifa=NULL;
        void * tmpAddrPtr=NULL;

        getifaddrs(&ifAddrStruct);

        for (ifa = ifAddrStruct; ifa != NULL; ifa = ifa->ifa_next) {
            Ipv4 auxIP;
            if (!ifa->ifa_addr) {
                continue;
            }
            if (ifa->ifa_addr->sa_family == AF_INET) { // check it is IP4
                // is a valid IP4 Address
                tmpAddrPtr=&((struct sockaddr_in *)ifa->ifa_addr)->sin_addr;
                char addressBuffer[INET_ADDRSTRLEN];
                inet_ntop(AF_INET, tmpAddrPtr, addressBuffer, INET_ADDRSTRLEN);
                printf("%s IP Address %s\n", ifa->ifa_name, addressBuffer);
                auxIP.setIpCharPtr(addressBuffer);
                interfaceC.push_back(auxIP);

            } else if (ifa->ifa_addr->sa_family == AF_INET6 && _ipv6) { // check it is IP6
                // is a valid IP6 Address
                tmpAddrPtr=&((struct sockaddr_in6 *)ifa->ifa_addr)->sin6_addr;
```

```
            char addressBuffer[INET6_ADDRSTRLEN];
            inet_ntop(AF_INET6, tmpAddrPtr, addressBuffer, INET6_ADDRSTRLEN);
            printf("%s IP Address %s\n", ifa->ifa_name, addressBuffer);
        }
    }
    if (ifAddrStruct!=NULL) freeifaddrs(ifAddrStruct);


    return interfaceC;

}
//
//  Interface.h
//  Route Request
//
//  Created by Pedro Loami on 29/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__Interface__
#define __Route_Request__Interface__

#include <stdio.h>
#include <sys/types.h>
#include <ifaddrs.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>

#include "Ipv4.h"
#include "string"

#include <iostream>
#include <vector>


class Interface{
    std::vector<Ipv4> ip_interface;
    std::vector<std::string> name_interface;

    Ipv4 netmask;
public:

    Interface();
```

```
    void setInterface();

    void setNetmask(Ipv4);

    Ipv4 getNetmask() const;

    std::vector<Ipv4> getIpInterface();

    Ipv4 getIpInterfaceByOrder(int n);

    std::string getNameInterfaceByOrder(int n);

    void addInterface(Ipv4 _newiP,Ipv4 _dstaddr);

    void rmInterface(Ipv4 _newiP, Ipv4 _mask,Ipv4 _dstaddr);

    void addRoute(Ipv4 _target,Ipv4 _gw);//, int _subinterface);

    void rmRoute(Ipv4 _ip, Ipv4 _gw, int _subinterface);

    int getSizeInterface()const;

    friend std::ostream &operator<<(std::ostream &, const Interface &);


};

#endif /* defined(__Route_Request__Interface__) */
//
//  Interface.cpp
//  Route Request
//
//  Created by Pedro Loami on 29/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "Interface.h"

Interface::Interface(){
    this->setInterface();
}
```

```
void Interface::setInterface(){
    std::vector<Ipv4> interfaceC;


    struct ifaddrs * ifAddrStruct=NULL;
    struct ifaddrs * ifa=NULL;
    void * tmpAddrPtr=NULL;
    void * tmpMaskPtr = NULL;

    getifaddrs(&ifAddrStruct);

    for (ifa = ifAddrStruct; ifa != NULL; ifa = ifa->ifa_next) {
        Ipv4 auxIP;
        if (!ifa->ifa_addr) {
            continue;
        }
        if (ifa->ifa_addr->sa_family == AF_INET) { // check it is IP4
            // is a valid IP4 Address
            tmpAddrPtr=&((struct sockaddr_in *)ifa->ifa_addr)->sin_addr;
            tmpMaskPtr=&((struct sockaddr_in *)ifa->ifa_netmask)->sin_addr;

            char maskBuffer[INET_ADDRSTRLEN];
            char addressBuffer[INET_ADDRSTRLEN];
            inet_ntop(AF_INET, tmpMaskPtr, maskBuffer, INET_ADDRSTRLEN );
            inet_ntop(AF_INET, tmpAddrPtr, addressBuffer, INET_ADDRSTRLEN);
            //printf("%s IP Address %s\n", ifa->ifa_name, addressBuffer);
            this->netmask.setIpCharPtr(maskBuffer);
            auxIP.setIpCharPtr(addressBuffer);
            this->ip_interface.push_back(auxIP);
            this->name_interface.push_back(ifa->ifa_name);


            //this->netmask.setIpCharPtr(inet_ntoa(<#struct in_addr#>);
            //inet_ntoa(((struct sockaddr_in *)&ifr_nmask.ifr_addr)->sin_addr)
            //this->netmask.setIpCharPtr(htons(ifa->ifa_netmask));
        } else if (ifa->ifa_addr->sa_family == AF_INET6 && false) { // check it is IP6   -> &&false
até implementar ipv6 na rede!
            // is a valid IP6 Address
            tmpAddrPtr=&((struct sockaddr_in6 *)ifa->ifa_addr)->sin6_addr;
            char addressBuffer[INET6_ADDRSTRLEN];
            inet_ntop(AF_INET6, tmpAddrPtr, addressBuffer, INET6_ADDRSTRLEN);
            //printf("%s IP Address %s\n", ifa->ifa_name, addressBuffer);
        }
    }
```

```
    if (ifAddrStruct!=NULL) freeifaddrs(ifAddrStruct);



}

void Interface::setNetmask(Ipv4 _mask){
    this->netmask = _mask;
}

Ipv4 Interface::getNetmask() const{
    return this->netmask;
}
std::vector<Ipv4> Interface::getIpInterface(){
    return ip_interface;
}

Ipv4 Interface::getIpInterfaceByOrder(int n){
    return ip_interface.at(n);
}

std::string Interface::getNameInterfaceByOrder(int n){
    return name_interface.at(n);
}

void Interface::addInterface(Ipv4 _newiP,Ipv4 _dstaddr){
    char buffer[50];
    sprintf(buffer,"ifconfig en1:%d %s netmask %s dstaddr %s",(int)this-
>ip_interface.size(),_newiP.getIpStr(),this->netmask.getIpStr(),_dstaddr.getIpStr());
    system(buffer);

    this->setInterface();

}

void Interface::rmInterface(Ipv4 _newiP, Ipv4 _mask,Ipv4 _dstaddr){
    //TODO: remover uma dada interface
}

void Interface::addRoute(Ipv4 _target,Ipv4 _gw){//, int _subinterface){
    char buffer[50],buffer2[20];
    std::cout<<"addrouteTest: "<<_target<<"<-x->"<<_gw<<std::endl;

    sprintf(buffer , "sudo route add -host %s ",_target.getIpStr());//, _gw.getIpStr());
```

```
    sprintf(buffer2, "-gateway %s",_gw.getIpStr());
    strcat(buffer, buffer2);
    std::cout<<buffer<<std::endl;
    system(buffer);
}

void Interface::rmRoute(Ipv4 _ip, Ipv4 _gw,int _subinterface){
    char buffer[50];
    sprintf(buffer , "route delete -host %s -gateway %s
en1:%d",_ip.getIpStr(),_gw.getIpStr(),_subinterface);
    system(buffer);
}

int Interface::getSizeInterface() const{
    return (int) this->ip_interface.size();
}

std::ostream &operator<<(std::ostream &output, const Interface & _interface){

    output<<"-_-_-_-_Interface_-_-_-_-"<<std::endl;
    std::cout<<"NetMask: "<<_interface.netmask<<std::endl;
    for(int j=0;j<_interface.ip_interface.size();j++){
        output<<_interface.name_interface.at(j)<<": "<<_interface.ip_interface.at(j)<<std::endl;
    }
    output<<"-_-_-_-_-_-_-_-_-_-_-_-_-"<<std::endl;

    return output;
}//
//  MessageMaintenance.h
//  Cliente UDP
//
//  Created by Pedro Loami on 10/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Cliente_UDP__MessageMaintenance__
#define __Cliente_UDP__MessageMaintenance__

#include "Ipv4.h"
#include <vector>

class MessageMaintenance{
public:
```

```
    virtual char *bufferSaida() = 0;

    virtual void lerBuffer(unsigned char *buffer) = 0;

    virtual unsigned char getType() const = 0;

    virtual unsigned char getDataLen() const = 0;

    virtual void setErrorType(int) = 0;

    virtual unsigned char getErrorType() const = 0;

    virtual void setErrorSourceAdd(Ipv4) = 0;

    virtual Ipv4 getErrorSourceAdd()const = 0;

    virtual void setErrorDestAdd(Ipv4) = 0;

    virtual Ipv4 getErrorDestAdd()const = 0;

    virtual void setDataLen(int n) = 0;

    virtual int getSalvage() const = 0;

    virtual void incSalvage() = 0;

    friend std::ostream& operator<<( std::ostream &os, const MessageMaintenance &b ){
        b.print(os);
        return os;
    }

    virtual void setUnreachableDestAdd(Ipv4) = 0;

    virtual Ipv4 getUnreachableDestAdd() const = 0;

    virtual void setUnsupportedOpt(unsigned char) = 0;

    virtual unsigned char getUnsupportedOpt() const = 0;

private:
        virtual void print( std::ostream& ) const =0;
};
#endif /* defined(__Cliente_UDP__MessageMaintenance__) */
//
```

```
//  MessageMaintenance.cpp
//  Cliente UDP
//
//  Created by Pedro Loami on 10/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "MessageMaintenance.h"
//
//  RERROR.h
//  Route Request
//
//  Created by Pedro Loami on 09/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__RERROR__
#define __Route_Request__RERROR__

#include "MessageMaintenance.h"
class RERROR: public MessageMaintenance {
    unsigned char dataLen;
    unsigned char erroType;
    unsigned char reservedMsalvage;

    Ipv4 errorSourceAdd;
    Ipv4 errorDestAdd;

    //campos não obrigatórios: depedem do tipo de erro
    Ipv4 unreachableDestAdd;
    unsigned char unsupportedOpt;
public:
    RERROR();

    char *bufferSaida();                    //virtual

    void lerBuffer(unsigned char *buffer);      //virtual

    unsigned char getType() const;              //virtual

    void print(std::ostream &) const;           //virtual

    unsigned char getDataLen() const;           //virtual
```

```
    void setErrorType(int);                    //virtual
    unsigned char getErrorType() const;        //virtual
    void setErrorSourceAdd(Ipv4);              //virtual
    Ipv4 getErrorSourceAdd()const;             //virtual
    void setErrorDestAdd(Ipv4);                //virtual
    Ipv4 getErrorDestAdd()const;               //virtual
    void setDataLen(int n);                    //virtual
    int getSalvage() const;                    //virtual
    void incSalvage();                         //virtual
    void setUnreachableDestAdd(Ipv4);          //virtual
    Ipv4 getUnreachableDestAdd() const;        //virtual
    void setUnsupportedOpt(unsigned char);     //virtual
    unsigned char getUnsupportedOpt() const;   //virtual
};


#endif /* defined(__Route_Request__RERROR__) */
//
//  RERROR.cpp
//  Route Request
//
//  Created by Pedro Loami on 09/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "RERROR.h"
RERROR::RERROR(){
    this->reservedMsalvage = 0x00;
    this->dataLen = 0x0A;
}
unsigned char RERROR::getType() const{
```

```
    return (unsigned char) 3;
}

char *RERROR::bufferSaida(){
    char * buffer;

    buffer = new char [(int)dataLen+1];

    buffer[0] = this->getType();
    buffer[1] = this->dataLen;
    buffer[2] = this->erroType;
    buffer[3] = this->reservedMsalvage;

    unsigned char *aux1 = errorSourceAdd.getIp();
    buffer[4] = aux1 [0];
    buffer[5] = aux1 [1];
    buffer[6] = aux1 [2];
    buffer[7] = aux1 [3];

    unsigned char *aux2 = errorDestAdd.getIp();
    buffer[8] = aux2 [0];
    buffer[9] = aux2 [1];
    buffer[10] = aux2 [2];
    buffer[11] = aux2 [3];

    switch (this->erroType) {
        case 1:{
            unsigned char *aux3 = this->getUnreachableDestAdd().getIp();
            buffer[12] = aux3 [0];
            buffer[13] = aux3 [1];
            buffer[14] = aux3 [2];
            buffer[15] = aux3 [3];

            break;
        }
        case 2:{
            break;
        }
        case 3:{
            buffer[12] = this->getUnsupportedOpt();
            break;
        }

        default:
```

```
            break;
        }

        return buffer;
}
void RERROR::lerBuffer(unsigned  char *buffer){
        if (buffer[0] != 3) {
            throw "ERROR! This is not a Route Error DSR message.";
        }
        this->dataLen = buffer[1];
        this->erroType = buffer[2];
        this->reservedMsalvage = buffer[3];

        this->errorSourceAdd.setIp(buffer[4], buffer[5], buffer[6], buffer[7]);

        this->errorDestAdd.setIp(buffer[8], buffer[9], buffer[10], buffer[11]);

        if(this->dataLen == 10){
            //Type-Specific Info is empty -> do nothing
        }
        else if (this->dataLen == 11){
            //Type-Specific Info is the Unsupported Option
            setUnsupportedOpt(buffer[12]);
        }
        else if (this->dataLen ==14){
            Ipv4 auxUnreachableAdd(buffer[12],buffer[13],buffer[14],buffer[15]);
            setUnreachableDestAdd(auxUnreachableAdd);
        }
        else{
            throw "ERROR! In lerBuffer the incomming DSR ERROR Maintenance Message has
invalid dataLen value.";
        }

}

void RERROR::print(std::ostream &os) const {
        os<<"Opt Type: "<<(unsigned int)this->getType()<<" <RERROR>"<<std::endl;
        switch (this->erroType) {
            case 1:{
                os<<"Error Type: "<<(unsigned int)this->erroType<<"
<NODE_UNREACHABLE>"<<std::endl;
                break;
            }
            case 2:{
```

```
            os<<"Error Type: "<<(unsigned int)this->erroType<<"
<FLOW_STATE_NOT_SUPPORTED>"<<std::endl;
            break;
        }
        case 3:{
            os<<"Error Type: "<<(unsigned int)this->erroType<<"
<OPTION_NOT_SUPORTED>"<<std::endl;
            break;
        }

        default:
            break;
    }



    os<<"dataLen: "<<(unsigned int)this->dataLen<<std::endl;
    os<<"Reseved & Salvage: "<<(unsigned int)this->reservedMsalvage<<std::endl;

    os<<"Erro Destination Add.: "<<this->errorDestAdd<<std::endl;

    os<<"Erro Source Add.: "<<this->errorSourceAdd<<std::endl;

    switch (this->erroType) {
        case 1:{
            os<<"Type-Specific Info: <Unreachabble Dest. Add.> "<<this-
>getUnreachableDestAdd()<<std::endl;
            break;
        }
        case 2:{
            os<<"Type-Specific Info: <EMPTY>"<<std::endl;
            break;
        }
        case 3:{
            os<<"Type-Specific Info: <Opt. Not Supported>"<<(unsigned int)this-
>getUnsupportedOpt()<<std::endl;
            break;
        }

        default:
            break;
    }
```

```
}
void RERROR::setErrorType(int i){
    switch (i) {
        case 1:{                    //NODE_UNREACHABLE
            this->erroType = 0x01;

            break;
        }
        case 2:{                    //FLOW_STATE_NOT_SUPPORTED
            this->erroType = 0x02;
            break;
        }

        case 3:{                    //OPTION_NOT_SUPPORTED
            this->erroType = 0x03;
            break;
        }

        default:{
            throw "ERROR! In 'void setErrorType(int)' input invalid";
            break;
        }
    }
}

unsigned char RERROR::getErrorType() const{
    return this->erroType;
}

void RERROR::setErrorSourceAdd(Ipv4 ip_error_source_add){
    this->errorSourceAdd = ip_error_source_add;
}

Ipv4 RERROR::getErrorSourceAdd()const{
    return errorSourceAdd;
}

void RERROR::setErrorDestAdd(Ipv4 ip_error_dest_add){
    this->errorDestAdd = ip_error_dest_add;
}

Ipv4 RERROR::getErrorDestAdd()const{
    return errorSourceAdd;
}
```

```
void RERROR::setDataLen(int n){
    this->dataLen = 0x0A + n*0x20;
}

unsigned char RERROR::getDataLen() const{
    return this->dataLen;
}

int RERROR::getSalvage() const{
    return this->reservedMsalvage;
}

void RERROR::incSalvage(){
    if (this->reservedMsalvage == 0x0F) {
        this->reservedMsalvage = 0x00;
    }
    else{
        this->reservedMsalvage += 0x01;
    }
}
void RERROR::setUnreachableDestAdd(Ipv4 _unreachableDestAdd){
    if(this->erroType == 1){
                this->unreachableDestAdd = _unreachableDestAdd;
                this->dataLen = 0x0E;

    }
    else{
        throw "ERROR! setUnreachableDestAdd called for a DSR Route Error of type different
than HOST UNREACHABLE (type 1)";
    }

}

Ipv4 RERROR::getUnreachableDestAdd() const{
    if(this->erroType == 1){
        return this->unreachableDestAdd;
    }
    else{
        throw "ERROR! getUnreachableDestAdd called for a DSR Route Error Message of type
different than HOST UNREACHABLE (type 1)";
    }
}
```

```
void RERROR::setUnsupportedOpt(unsigned char _unsupportedOpt){
    if(this->erroType == 3){
        this->unsupportedOpt = _unsupportedOpt;
        this->dataLen = 0x0B;
    }
    else{
        throw "ERROR! setUnsupportedOpt called for a DSR Route Error of type different than
OPTION_NOT_SUPPORTED(type 3)";
    }
}

unsigned char RERROR::getUnsupportedOpt() const{
    if(this->erroType == 3){
        return this->unsupportedOpt;
    }
    else{
        throw "ERROR! getUnsupportedOpt called for a DSR Route Error Message of type
different than OPTION_NOT_SUPPORTED(type 3)";
    }

}//
//  Route.h
//  Route Request
//
//  Created by Pedro Loami on 15/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__Route__
#define __Route_Request__Route__

#include <vector>
#include "Ipv4.h"

class Route {

    std::vector<Ipv4> intermed;
    Ipv4 destination;

public:
    Route(Ipv4 destination, std::vector<Ipv4> _intermed);
    Route();
    Ipv4 getDest() const;
```

```
    std::vector<Ipv4> getIntermed() const;
    void setRoute(Ipv4 _destination, std::vector<Ipv4> _intermed);
    std::vector<Ipv4> appendInter(std::vector<Ipv4>);
    void printRoute(int)const;

    bool operator == (const Route &)const;
    bool operator != (const Route &route)const{
        return !(*this == route);
    }


};

#endif /* defined(__Route_Request__Route__) */
//
//  Route.cpp
//  Route Request
//
//  Created by Pedro Loami on 15/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "Route.h"

Route::Route(Ipv4 destination, std::vector<Ipv4> _intermed): destination(destination){
    this->intermed=_intermed;
}

Route::Route(){
    this->destination.setIp(0, 0, 0, 0);

}


Ipv4 Route::getDest()const{
    return this->destination;
}

std::vector<Ipv4> Route::getIntermed() const{
    return this->intermed;
}

void Route::setRoute(Ipv4 _destination, std::vector<Ipv4> _intermed){
    destination = _destination;
```

```
    intermed = _intermed;

}
std::vector<Ipv4> Route::appendInter(std::vector<Ipv4> _intermed){
    std::vector<Ipv4> _novo;
    _novo.reserve(_intermed.size() + this->intermed.size());
    _novo.insert(_novo.end(), _intermed.begin(),_intermed.end());
    _novo.insert(_novo.end(), this->intermed.begin(),this->intermed.end());

    return _novo;
}
bool Route:: operator== (const Route &route)const{
    bool aux = true;

    if(!(this->destination == route.destination)){
        aux = false;
    }

    if(this->intermed != route.intermed){
        aux=false;
    }

    return aux;
}
void Route::printRoute(int _i)const{
    std::cout<<"*********Route:"<<_i<<"*********"<<std::endl;
    std::cout<<"dest:"<<destination<<std::endl;
    int cont=0;
    for (std::vector<Ipv4>::const_iterator it = this->intermed.begin(); it != this->intermed.end();
++it){

    std::cout<<"intermed["<<++cont<<"]="<<*it<<std::endl;

    }


}


//
//  RReply.h
//  Cliente UDP
//
//  Created by Pedro Loami on 09/03/15.
```

```
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Cliente_UDP__RReply__
#define __Cliente_UDP__RReply__

#include <vector>
#include "Message.h"

class RReply : public Message{
    unsigned char dataLen;              //4n+1
    bool L;                             //Last Hop External = 0x00 ou 0x80
    std::vector<Ipv4> intermed;

public:
    RReply(std::vector<Ipv4> inter, bool _L);

    RReply();

    char *bufferSaida();                        //virtual

    char *bufferSaidaPiggy(char * _buffer);     //virtual

    void lerBuffer(unsigned char *buffer);      //virtual

    unsigned char getType() const;              //virtual

    void setL(bool);                            //virtual

    int getL() const;                           //virtual

    void addNodeIntermed(Ipv4);                 //virtual

    void print(std::ostream &) const;           //virtual

    void setTarget(Ipv4);                       //virtual

    Ipv4 getTarget() const;                     //virtual

    void setId(unsigned char *);                //virtual

    void setId(u_int16_t &);                    //virtual

    unsigned char * getId() const;              //virtual
```

```
    uint16_t  getNumericalId() const;           //virtual

    void setIntermed(std::vector<Ipv4> _inter);  //virtual

    void setDataLen();                          //virtual

    unsigned char getDataLen() const;           //virtual

    std::vector<Ipv4> getIntermed() const;      //virtual



};
#endif /* defined(__Cliente_UDP__RReply__) */
//
//  RReply.cpp
//  Cliente UDP
//
//  Created by Pedro Loami on 09/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include <iostream>
#include <fstream>

#include "RReply.h"


RReply::RReply(std::vector<Ipv4> inter, bool _L){
    L=_L;
    setIntermed(inter);
}

RReply::RReply(){
    this->L = false;
    dataLen = 1;
}

char *RReply::bufferSaida(){
    int cont=3;

    static char * buffer;
```

```
    buffer = new char [(int)dataLen+2];

    buffer[0] = getType();
    buffer[1] = dataLen;

    if(L){
        buffer[2] = 0x80;
    }
    else{
        buffer[2] = 0;
    }

    for (std::vector<Ipv4>::iterator it = this->intermed.begin(); it != this->intermed.end(); ++it){
        unsigned char *aux;
        aux = it->getIp();
        buffer[cont++] = aux[0];
        buffer[cont++] = aux[1];
        buffer[cont++] = aux[2];
        buffer[cont++] = aux[3];
    }
    return buffer;
}

char * RReply::bufferSaidaPiggy(char * _buffer){
    char * erro = "Error! This class doesn't implemment this method. RReply::bufferSaidaPiggy";
    return erro;
}

void RReply::lerBuffer(unsigned char *buffer){
    int cont=3;
    Ipv4 auxIp;
    if (buffer[0] != 2) {
        throw "Tipo inválido.";
    }
    dataLen = buffer[1];
    L = buffer[2] & 0x80;

    intermed.clear();
    for(int i = 0; i<((int)this->dataLen-1)/4; i++){
        auxIp.setIp(buffer[cont],buffer[cont+1],buffer[cont+2],buffer[cont+3]);
        intermed.push_back(auxIp);
        cont+=4;
    }
}
```

```
unsigned char RReply::getType() const{
    return (unsigned char) 2;
}

void RReply::setL(bool _L){
    this->L=_L;
}

int RReply::getL() const{
    return L;
}

void RReply::addNodeIntermed(Ipv4 _hopIp){
    intermed.push_back(_hopIp);
    dataLen = 4*this->intermed.size()+1;
}

void RReply::print(std::ostream &os) const{
    os<<"type:"<<(unsigned int)this->getType()<<"<RReply>"<<std::endl;
    os<<"dataLen:"<<(unsigned int)this->dataLen<<std::endl;
    os<<"n:"<<this->intermed.size()<<std::endl;
    int cont = 0;
    std::vector<Ipv4> _intermed = this->intermed;
    for (std::vector<Ipv4>::iterator it = _intermed.begin(); it != _intermed.end(); ++it, cont++){
        os<<"intermed["<<cont<<"]:"<< *it << std::endl;
    }
}

void RReply::setTarget(Ipv4 _target){}


Ipv4 RReply::getTarget() const{
    Ipv4 VOID;
    return VOID;
}

void RReply::setId(unsigned char *){}

void RReply::setId(u_int16_t & ){}


unsigned char * RReply::getId() const{
    throw "Error the class RReply does not implement this method";
```

```
}

uint16_t  RReply::getNumericalId() const{
    throw "ERRO!";
}

void RReply::setIntermed(std::vector<Ipv4> _inter){
    this->intermed = _inter;
    dataLen = 4*this->intermed.size()+1;
}

void RReply::setDataLen(){

    this->dataLen = 4*intermed.size()+1;
}

unsigned char RReply::getDataLen() const{
    return this->dataLen;
}

std::vector<Ipv4> RReply::getIntermed() const{
    return this->intermed;
}

//
//  RReq.h
//  Route Request
//
//  Created by Pedro Loami on 01/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__RReq__
#define __Route_Request__RReq__


//#include <stdio.h>
#include <vector>
#include "Message.h"

class RReq : public Message {
    Ipv4 targetIPaddr;                      // Destination IP address
    unsigned char dataLen;                      // Length of the message. Equals to 4*n+6, where n
is the number of IP addresses in the path
```

```
    unsigned char idNumber[2];                  // Identification number: unique value generated by
the initiator
    std::vector<Ipv4> intermed;                 // Address[1..n]: path

public:

    RReq();

    RReq(Ipv4 target,std::vector<Ipv4> inter,u_int16_t &);

    char *bufferSaida();                        //virtual

    char *bufferSaidaPiggy(char * _buffer);

    void lerBuffer(unsigned char *buffer);      //virtual

    unsigned char getType() const;              //virtual

    void setL(bool);                            //virtual

    int getL() const;                           //virtual

    void addNodeIntermed(Ipv4);                 //virtual

    void print(std::ostream &) const;

    void setTarget(Ipv4);                       //virtual

    Ipv4 getTarget() const;                     //virtual

    void setId(unsigned char *);                //virtual

    void setId(u_int16_t &);

    unsigned char * getId() const;              //virtual

    void setIntermed(std::vector<Ipv4> _inter);  //virtual

    void setDataLen();                          //virtual

    unsigned char getDataLen() const;           //virtual

    //unsigned char getId(int);                 //virtual
```

```
    uint16_t  getNumericalId() const;           //virtual

    std::vector<Ipv4> getIntermed() const;      //virtual


//    void print(std::ostream &) const;



};


#endif /* defined(__Route_Request__RReq__) */
//
//  RReq.cpp
//  Route Request
//
//  Created by Pedro Loami on 01/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//
#include "RReq.h"

RReq::RReq(Ipv4 target,std::vector<Ipv4> inter, u_int16_t &_gem){
    dataLen = 4*inter.size()+1;
    targetIPaddr.setIpVec(target.getIp());
    this->setIntermed(inter);

    this->idNumber[1] = _gem & 0xFF;
    this->idNumber[0] = _gem >> 8;
    _gem++;
}

RReq::RReq(){
    dataLen = 6;
    idNumber[0] = 0;
    idNumber[1] = 0;
}

char *RReq::bufferSaida(){
    char * buffer;
    int cont=8;
    buffer = new char [(int)dataLen+1];

    buffer[0] = getType();
```

```
    buffer[1] = dataLen;
    buffer[2] = idNumber[0];
    buffer[3] = idNumber[1];

    unsigned char *aux = targetIPaddr.getIp();
    buffer[4] = aux [0];
    buffer[5] = aux [1];
    buffer[6] = aux [2];
    buffer[7] = aux [3];

    for (std::vector<Ipv4>::iterator it = this->intermed.begin(); it != this->intermed.end(); ++it){
        unsigned char *aux2;
        aux2 = it->getIp();

        buffer[cont++] = aux2[0];
        buffer[cont++] = aux2[1];
        buffer[cont++] = aux2[2];
        buffer[cont++] = aux2[3];
    }
    //    buffer[cont] = '\0';
    return buffer;
}

char * RReq::bufferSaidaPiggy(char * _buffer){
    char * buffer;
    unsigned char _dataLen = _buffer[1];
    buffer = new char [(int)_dataLen+8];

    buffer[0] = getType();
    buffer[1] = dataLen;
    buffer[2] = idNumber[0];
    buffer[3] = idNumber[1];

    unsigned char *aux = targetIPaddr.getIp();
    buffer[4] = aux [0];
    buffer[5] = aux [1];
    buffer[6] = aux [2];
    buffer[7] = aux [3];

    for(int j = 0; j<(int)_dataLen;j++){
        buffer[j+8] = _buffer[j];
    }

    return buffer;
```

```
}

void RReq::lerBuffer(unsigned char *buffer){
    int cont=8;
    Ipv4 auxIp;
    if (buffer[0] != 1) {
        throw "Tipo inválido.";
    }
    dataLen = buffer[1];
    idNumber[0] = buffer[2];
    idNumber[1] = buffer[3];

    targetIPaddr.setIp(buffer[4], buffer[5], buffer[6], buffer[7]);

    intermed.clear();
    for(int i =0; i<((int)this->dataLen-6)/4; i++){
        auxIp.setIp(buffer[cont],buffer[cont+1],buffer[cont+2],buffer[cont+3]);
        intermed.push_back(auxIp);
        cont+=4;
    }
}

unsigned char RReq::getType() const{
    return (unsigned char) 1;
}

void RReq::setTarget(Ipv4 target){
    targetIPaddr.setIpVec(target.getIp());
}

Ipv4 RReq::getTarget() const{
    return this->targetIPaddr;
}

void RReq::setId(unsigned char *input){
    idNumber[0] = input[0];
    idNumber[1] = input[1];

}

void RReq::setId(u_int16_t &_gem){
    this->idNumber[1] = _gem & 0xFF;
    this->idNumber[0] = _gem >> 8;
```

```
    _gem++;
}

unsigned char * RReq::getId()const{
    static unsigned char * aux;
    aux = new unsigned char[2]();
    aux[0] = this->idNumber[0];
    aux[1] = this->idNumber[1];

    return aux;
}

uint16_t RReq::getNumericalId() const{
    return  idNumber[0]<<8 | idNumber[1];
}

void RReq::setIntermed(std::vector<Ipv4> _inter){
    this->intermed = _inter;
    dataLen = 4*this->intermed.size()+6;
}

std::vector<Ipv4> RReq::getIntermed() const{
    return this->intermed;
}

void RReq::addNodeIntermed(Ipv4 _hopIp){
    intermed.push_back(_hopIp);
    dataLen = 4*this->intermed.size()+6;
}

void RReq::setL(bool _L){}

int RReq::getL() const{
    throw "RReq does not implement this method";
}

void RReq::setDataLen(){
    this->dataLen = 4*intermed.size()+6;
}

unsigned char RReq::getDataLen() const{
    return this->dataLen;
}
```

```
//unsigned char RReq::getId(<#int#>)

void RReq::print(std::ostream &os) const {
    os<<"type: "<<(unsigned int)this->getType()<<"<RReq>"<<std::endl;
    os<<"dataLen: "<<(unsigned int)this->dataLen<<std::endl;
    os<<"idNumber: "<<(unsigned int)this->idNumber[0]<<(unsigned int)this-
>idNumber[1]<<std::endl;
    os<<"targetIp: "<<this->targetIPaddr<<std::endl;
    os<<"n:"<<this->intermed.size()<<std::endl;
    int cont = 0;
    std::vector<Ipv4> _intermed = this->intermed;
    for (std::vector<Ipv4>::iterator it = _intermed.begin(); it != _intermed.end(); ++it, cont++){
        os<<"intermed["<<cont<<"]: "<< *it <<std::endl;
    }
}

//
//  RReqTable.h
//  Route Request
//
//  Created by Pedro Loami on 22/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__RReqTableType__
#define __Route_Request__RReqTableType__

#include "Ipv4.h"
#include <ctime>


class RReqTableType{
    unsigned char idNumber[2];
    Ipv4 target;
//    time_t timeRReq;
public:
    RReqTableType();

    RReqTableType(unsigned char,unsigned char,Ipv4);

    unsigned char  getIdNumber(int _i) const;

    void setIdNumber(unsigned char,unsigned char);
```

```
    Ipv4 getTarget() const;

    void setTarget(Ipv4);

//    void setTime();

//    time_t getTime() const;

    void registerRReq(unsigned char,unsigned char,Ipv4);

    friend std::ostream &operator<<(std::ostream &, const RReqTableType &);

    bool operator == (const RReqTableType &) const;

    bool operator != (const RReqTableType & right) const{
        return !(*this == right);
    }


};
#endif /* defined(__Route_Request__RReqTable__) */
//
//  RReqTable.h
//  Route Request
//
//  Created by Pedro Loami on 22/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Route_Request__RReqTable__
#define __Route_Request__RReqTable__

#define TIMEOUT 15000

#include <chrono>
#include <thread>
#include <vector>
#include <mutex>
#include <algorithm>    //para o linux
#include "RReqTableType.h"

#define RREQTIME  7   //s
```

```
class RReqTable{

    std::vector<RReqTableType> Reqtable;

public:
    void appendRReq(RReqTableType _rreqType);

    bool isThere(RReqTableType _rreqType);

    bool isValid(RReqTableType _rreType);

    friend std::ostream &operator<<(std::ostream &, const RReqTable &);

    void teste (int);
    static void deleteEntry(  RReqTable *rt, RReqTableType r);

    //void onlyDelete(RReqTable *, std::vector<RReqTableType>::iterator);

};

#endif /* defined(__Route_Request__RReqTable__) */
//
//  RReqTable.cpp
//  Route Request
//
//  Created by Pedro Loami on 22/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//



#include "RReqTable.h"

std::mutex g_lock;


void RReqTable::appendRReq(RReqTableType _rreqType){
    if( !isThere(_rreqType)){
        this->Reqtable.push_back(_rreqType);
        std::cout << "Inserting RReq " << _rreqType << " in table." << std::endl;
        std::thread (deleteEntry, this, _rreqType).detach();
    }
```

```
}


bool RReqTable::isThere(RReqTableType _rreqType){
    std::vector<RReqTableType>::iterator it;
    it = find (this->Reqtable.begin(),this->Reqtable.end(),_rreqType);
    if(it != Reqtable.end()){
        return true;
    }
    else{
        return false;
    }


}
std::ostream &operator<<(std::ostream &output, const RReqTable & _table){

    output<<"RReq Table:"<<std::endl;

    for (std::vector<RReqTableType>::const_iterator it = _table.Reqtable.begin(); it !=
_table.Reqtable.end(); ++it){

        std::cout<<*it;
    }//for


    return output;
}

//
//  SocketRaw.h
//  Socket Raw
//
//  Created by Bianca Lodoli on 09/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Socket_Raw__SocketRaw__
#define __Socket_Raw__SocketRaw__


#define PCKT_LEN 8192

#include <stdio.h>
```

```
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <string.h>
#include <strings.h>

#include <netinet/ip.h>
#include <netinet/udp.h>


#include "Ipv4.h"

class SocketRaw {
    int raw_socket;
    struct ip *ipHeader;
    struct udphdr *udpHeader;
    char *message;
    struct sockaddr_in source_addr, dest_addr;
    unsigned short csum ( unsigned short *buf, int nwords );
public:
    SocketRaw(Ipv4 ip_origin, int port_origin, Ipv4 ip_dest, int port_dest);
    long int enviar(char *message, int dataLen);
    ~SocketRaw();
};

#endif /* defined(__Socket_Raw__SocketRaw__) */
//
//  SocketRaw.cpp
//  Socket Raw
//
//  Created by Bianca Lodoli on 09/03/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "SocketRaw.h"


SocketRaw::SocketRaw(Ipv4 ip_origin, int port_origin, Ipv4 ip_dest, int port_dest){
```

```
static char *buffer = new char[PCKT_LEN];

this->ipHeader = (struct ip *)buffer;
this->udpHeader = (struct udphdr *) (buffer + sizeof(struct ip));

this->message= (char *) (buffer + sizeof(struct ip) +sizeof(struct udphdr));

int one = 1;
const int *val = &one;

memset(buffer, 0, PCKT_LEN);

raw_socket = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);

if(raw_socket < 0) {
    perror("socket() error");
    throw "socket() error";
}


int broadcast=1;
if (setsockopt(raw_socket,SOL_SOCKET,SO_BROADCAST,
            &broadcast,sizeof(broadcast))==-1) {
        throw "Erro no setsockopt.";
}




this->source_addr.sin_family = AF_INET;
this->dest_addr.sin_family = AF_INET;

this->source_addr.sin_port = htons(port_origin);
this->dest_addr.sin_port = htons(port_dest);

this->source_addr.sin_addr.s_addr = inet_addr(ip_origin.getIpStr());
this->dest_addr.sin_addr.s_addr = inet_addr(ip_dest.getIpStr());

ipHeader->ip_hl = 5;
ipHeader->ip_v = 4;
ipHeader->ip_tos = 16;
ipHeader->ip_len = sizeof(struct ip) + sizeof(struct udphdr);
ipHeader->ip_id = htons(54321);
ipHeader->ip_ttl = 64;
```

```
    ipHeader->ip_p = 17;     //protocolo: udp = 17 dsr = 48
    ipHeader->ip_src = source_addr.sin_addr;
    ipHeader->ip_dst = dest_addr.sin_addr;

    udpHeader->uh_sport = htons(port_origin);
    udpHeader->uh_ulen = htons(sizeof(struct udphdr));
    udpHeader->uh_dport = htons(port_dest);

    ipHeader->ip_sum = csum((unsigned short *)buffer, sizeof(struct ip) + sizeof(struct udphdr));

    if(setsockopt(raw_socket, IPPROTO_IP, IP_HDRINCL, val, sizeof(one)) < 0) {
        throw "setsockopt() error";
    }

}

long int SocketRaw::enviar(char *message, int dataLen){
    long int i;
    char *buffer = (char *)this->ipHeader;
    this->ipHeader->ip_len = sizeof(struct ip) + sizeof(struct udphdr) + dataLen;
    this->udpHeader->uh_ulen = htons(sizeof(struct udphdr) + dataLen);
    memcpy(this->message, message, dataLen);
    i = sendto(raw_socket, buffer, ipHeader->ip_len, 0, (struct sockaddr *)&dest_addr,
sizeof(dest_addr));
    if(i < 0){
//        perror("sendto() error");
        throw "sendto() error";
    }
    return i;
}

unsigned short SocketRaw::csum(unsigned short *buf, int nwords){
    unsigned long sum;
    for(sum=0; nwords>0; nwords--)
        sum += *buf++;
    sum = (sum >> 16) + (sum &0xffff);
    sum += (sum >> 16);
    return (unsigned short)(~sum);
}

SocketRaw::~SocketRaw(){
    close(raw_socket);
}//
//  SourceRouteOpt.h
```

```
//  Cliente UDP
//
//  Created by Pedro Loami on 11/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#ifndef __Cliente_UDP__SourceRouteOpt__
#define __Cliente_UDP__SourceRouteOpt__

#include "MessageMaintenance.h"
class SourceRouteOpt{
    uint8_t dataLen;                    //4*n+2
    bool F;
    bool L;

    uint8_t salvage;
    uint8_t segsLeft;

    std::vector<Ipv4> address;          // com Size  n
public:

    SourceRouteOpt();

    char *bufferSaida();

    void lerBuffer(unsigned char *buffer);



    uint8_t getType() const;

    uint8_t getDataLen() const;

    bool getF() const;

    void setF(bool);

    bool getL() const;

    void setL(bool);

    uint8_t getSalvage() const;

    void setSalvage(uint8_t);
```

```
    uint8_t getSegsLeft()const;

    void setSegsLeft(uint8_t);

    void addNodeIntermed(Ipv4);

    void setIntermed(std::vector<Ipv4> _inter);

    std::vector<Ipv4> getIntermed() const;


    friend std::ostream &operator<<(std::ostream &, const SourceRouteOpt &);


};

#endif /* defined(__Cliente_UDP__SourceRouteOpt__) */
//
//  SourceRouteOpt.cpp
//  Cliente UDP
//
//  Created by Pedro Loami on 11/05/15.
//  Copyright (c) 2015 IME. All rights reserved.
//

#include "SourceRouteOpt.h"

SourceRouteOpt::SourceRouteOpt(){

    this->dataLen = 2;
    this->F = 0;
    this->L = 0;
    this->salvage = 0;
    this->segsLeft = 0;
}
//
char * SourceRouteOpt::bufferSaida(){
    char * buffer;
    int cont=4;
    buffer = new char [(int)dataLen+2];

    buffer[0] = getType();
    buffer[1] = dataLen;
```

```
    uint8_t byte1=0;
    if(this->F == true){        //bit 7
        byte1 += 128;
    }
    if(this->L == true){       //bit 6
        byte1 += 64;
    }
                            //bits 2-5 são nulos
    byte1 += (this->salvage & 0x0C)>>2;      //bits 1 e 0
    buffer[2] = byte1;


    uint8_t byte2 = 0;
    byte2 = (this->salvage & 0x03)<<6;
    byte2 += this->segsLeft;
    buffer[3] = byte2;

    for (std::vector<Ipv4>::iterator it = this->address.begin(); it != this->address.end(); ++it){
        unsigned char *aux2;
        aux2 = it->getIp();

        buffer[cont++] = aux2[0];
        buffer[cont++] = aux2[1];
        buffer[cont++] = aux2[2];
        buffer[cont++] = aux2[3];
    }

    return buffer;
}

void SourceRouteOpt::lerBuffer(unsigned char *buffer){
    int cont=4;
    Ipv4 auxIp;

    if (buffer[0] != 96) {
        throw "ERROR! This is not a DSR Source Route Opt message.";
    }

    this->dataLen = buffer[1];

    this->F = buffer[2] & 0x80;

    this->L = buffer[2] & 0x40;
```

```
    this->salvage = ((buffer[2] & 0x03)<<2) | ((buffer[3] & 0xC0)>>6);

    this->segsLeft = buffer[3] & 0x3F;

    address.clear();

    for(int i =0; i<((int)this->dataLen-2)/4; i++){
        auxIp.setIp(buffer[cont],buffer[cont+1],buffer[cont+2],buffer[cont+3]);
        address.push_back(auxIp);
        cont+=4;
    }
}


//
uint8_t SourceRouteOpt::getType() const{
    return 96;
}

uint8_t SourceRouteOpt::getDataLen() const{
    return this->dataLen;
}

bool SourceRouteOpt::getF() const{
    return this->F;
}

void SourceRouteOpt::setF(bool _F){
    this->F=_F;
}

bool SourceRouteOpt::getL() const{
    return this->L;
}

void SourceRouteOpt::setL(bool _L){
    this->L=_L;
}

uint8_t SourceRouteOpt::getSalvage() const{
    return this->salvage;
}
```

```
void SourceRouteOpt::setSalvage(uint8_t _s){
    if(_s > 15){
        throw "ERROR! Exceeded maximum size of DSR Source Route Option Salvage(4 bits).";
    }
    this->salvage = _s;
}

uint8_t SourceRouteOpt::getSegsLeft() const{
    return this->segsLeft;
}

void SourceRouteOpt::setSegsLeft(uint8_t _s){
    if(_s > 63){
        throw "ERROR! Exceeded maximum size of DSR Source Route Option Salvage(6 bits).";
    }
    this->segsLeft = _s;
}

void SourceRouteOpt::addNodeIntermed(Ipv4 _hopIp){
    this->address.push_back(_hopIp);
    this->dataLen = 4*this->address.size()+2;
}

void SourceRouteOpt::setIntermed(std::vector<Ipv4> _inter){
    this->address = _inter;
    this->dataLen = 4*this->address.size()+2;
}

std::vector<Ipv4> SourceRouteOpt::getIntermed() const{
    return this->address;
}


std::ostream &operator<<(std::ostream &os, const SourceRouteOpt &objSoourceRouteOptRef){

    os<<"Opt Type: "<<(unsigned int)objSoourceRouteOptRef.getType()<<" <Source Route
Opt>"<<std::endl;

    os<<"dataLen: "<<(unsigned int)objSoourceRouteOptRef.dataLen<<std::endl;

    os<<"First Hop External (F): "<<objSoourceRouteOptRef.F<<std::endl;

    os<<"Last Hop External (L): "<<objSoourceRouteOptRef.L<<std::endl;
```

```
    os<<"Salvage: "<<(unsigned int)objSoourceRouteOptRef.salvage<<std::endl;

    os<<"Segments Left: "<<(unsigned int)objSoourceRouteOptRef.segsLeft<<std::endl;

    os<<"n: "<<objSoourceRouteOptRef.address.size()<<std::endl;

    int cont = 0;

    std::vector<Ipv4> _address = objSoourceRouteOptRef.address;

    for (std::vector<Ipv4>::iterator it = _address.begin(); it != _address.end(); ++it, cont++){
        os<<"adress["<<cont<<"]:"<< *it <<std::endl;
    }

    return os;
}
```