Alexandre Gava Menezes

INTERCONECTANDO SENSORES COM TRANSPARÊNCIA NUMA REDE DE SENSORES SEM FIO SEGURA DIVIDIDA EM CLUSTERS

Florianópolis
2007

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Alexandre Gava Menezes

INTERCONECTANDO SENSORES COM TRANSPARÊNCIA NUMA REDE DE SENSORES SEM FIO SEGURA DIVIDIDA EM CLUSTERS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Carlos Becker Westphall
Orientador

Florianópolis, Fevereiro de 2007

INTERCONECTANDO SENSORES COM TRANSPARÊNCIA NUMA REDE DE SENSORES SEM FIO SEGURA DIVIDIDA EM CLUSTERS

Alexandre Gava Menezes

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, Área de Concentração Redes de Computadores e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____
Prof. Raul Sidnei Wazlawick, Dr.
Coordenador

Banca Examinadora:

_____
Prof. Carlos Becker Westphall, Dr.
Orientador

_____
Profa. Carla Merkle Westphall, Dra.

_____
Prof. Mário Antônio Ribeiro Dantas, Dr.

_____
Prof. Rômulo Silva de Oliveira, Dr.

Dedico este trabalho aos meus pais Tito Lívio e Jurema, e a minha esposa Elaine pelo amor, compreensão, carinho e incentivo recebidos durante todo o processo de desenvolvimento do mesmo.

AGRADECIMENTOS

Agradeço a Deus por tudo, em especial por Seu infinito amor para comigo. Sem Ele, nada seria possível.

Agradeço aos meus pais por todo o esforço despendido ao longo da minha vida para que culminasse na realização deste trabalho.

Agradeço a minha esposa Elaine, pela exemplar companheira que ela é. Seu amor, apoio, carinho e compreensão foram definitivos. Sem ela, este trabalho não seria o mesmo.

Agradeço aos meus irmãos e familiares pelo apoio e compreensão necessários para a realização deste trabalho.

Agradeço ao meu orientador Prof. Dr. Carlos Becker Westphall pela oportunidade, apoio, amizade e confiança depositados em mim e no meu trabalho.

Agradeço a Universidade Federal de Santa Catarina, o Departamento de Informática e de Estatística e em especial o Núcleo de Processamento de Dados, representado pelo grande amigo e diretor Nicolau Jorge Haviaras, pelo apoio oferecido para a realização desta pesquisa.

Agradeço em especial a amiga Káthia Regina Jucá pelas críticas e conselhos dados, desempenhando um papel de co-orientadora, que foi de suma importância.

Agradeço em especial também ao amigo Gustavo Adolpho Rangel Monteiro, pelas inúmeras horas dedicadas na tentativa de sempre ajudar no aperfeiçoamento deste trabalho.

Agradeço aos colegas: André Zimmermann, David Benjamin Silver, Edison Tadeu Lopes Melo, Fabrício Hipólito, Fausto Vetter, Guilherme Arthur Jerônimo, Guilherme Cordeiro, Guilherme Eliseu Rhoden, João Batista Furtuoso, José Marcos da Silva, Marcelo Cabral de Souza, Márcio Clemes e Murilo Vetter, que de uma forma ou de outra ajudaram em muito, a tornar esta pesquisa possível.

Agradeço ao professor Dr. Dalton Francisco de Andrade por seu precioso tempo despendido em ajudar a compreender e a abordar melhor o ambiente das simulações.

Agradeço ao professor Dr. Walter Pereira Carpes Júnior pelas explicações dadas a respeito da propagação de sinais no espaço.

Agradeço ao Eng. Pedro Vale Estrela, pela ajuda em configurar o simulador NS-2.

Agradeço os professores membros da banca, Dra. Carla Merkle Westphall, Dr. Mário Antônio Ribeiro Dantas e Dr. Rômulo Silva de Oliveira pelas contribuições e orientações que vieram a enriquecer mais o trabalho.

SUMÁRIO

SUMÁRIO VII

LISTA DE ILUSTRAÇÕES X

LISTA DE TABELAS XIII

LISTA DE ABREVIAÇÕES XIV

RESUMO XVI

ABSTRACT XVII

1. INTRODUÇÃO 1

1.1. Objetivos 2

1.2. Metodologia 3

1.3. Justificativa e resultados esperados 7

1.4. Limitações 8

1.5. Estrutura da dissertação 8

2. SEGURANÇA EM AMBIENTES DISTRIBUÍDOS 10

2.1. Definições importantes 10

2.1.1. Requisitos de Serviços Seguros e Ataques de Segurança 10

2.1.2. Criptografia 12

2.1.2.1. Criptografia simétrica 13

2.1.2.2. Criptografia assimétrica 13

2.1.2.3. Funções resumo (*hash*) 16

2.1.2.4. Assinatura digital 17

2.1.2.5. Troca de Chaves de Diffie-Hellman 18

2.1.2.6. Infraestrutura de chaves públicas - ICP 18

2.1.3. Gerenciamento de chaves 19

3. REDES WIRELESS 21

3.1. Roteamento em redes ad hoc 21

3.1.1. Redes ad hoc 21

3.1.2. Roteamento pró-ativo e reativo 22

3.1.3. O protocolo AODV 23

3.1.4. Ataques possíveis e falhas de segurança do protocolo AODV 26

3.1.5. As extensões do SAODV 28

3.2. Trabalhos correlatos 29

4. PROPOSTA DE PROTOCOLO HÍBRIDO 33

4.1. Disposição dos sensores e dos clusters 33

4.2. O esquema de distribuição de chaves 35

4.2.1. O protocolo de distribuição de chaves 36

4.2.2. Protegendo e controlando a redistribuição das chaves de grupo 40

4.3. Interconectando Clusters com Transparência 41

4.3.1. O protocolo de roteamento 42

4.3.1.1. Adequando o protocolo AODV e as extensões SAODV para clusters 42

4.3.1.2. O processo de estabelecimento de rotas de comunicação 44

5. RESULTADOS EXPERIMENTAIS 51

5.1. O *Network Simulator* 2 – NS2 51

5.2. O modelo de sensor utilizado 52

5.3. A disposição dos sensores em um cluster 53

5.4. A disposição dos clusters 53

5.5. Alterações necessárias no NS-2 54

5.6. As simulações realizadas 55

5.6.1. Os valores utilizados 55

5.6.2. O ambiente simulado 56

5.6.2.1. A montagem dos cenários no simulador..... 57

5.7. Os resultados obtidos.....60

6. CONCLUSÕES E TRABALHOS FUTUROS.....72

REFERÊNCIAS BIBLIOGRÁFICAS.....74

ANEXO I.....77

ANEXO II.....82

ANEXO III.....116

ANEXO IV 120

LISTA DE ILUSTRAÇÕES

Diagrama 1: Trajetória da pesquisa 4

Diagrama 2: Desenvolvimento da solução 5

Diagrama 3: As simulações 6

Figura 1: Os quatro tipos de ataques de segurança 11

Figura 2: Criptografia com chave simétrica 13

Figura 3: Uso de criptografia assimétrica para garantir confidencialidade 14

Figura 4: Uso de criptografia assimétrica para garantir autenticidade 14

Figura 5: Uma função *hash* aplicada sobre um texto plano 16

Figura 6: Criação de uma assinatura digital 17

Figura 7: Verificação de uma assinatura digital 18

Figura 8: Comunicação direta numa rede ad hoc 21

Figura 9: Comunicação de múltiplos saltos numa rede ad hoc 22

Figura 10: O formato de uma mensagem de requisição de rota RREQ do AODV 24

Figura 11: O formato da mensagem de resposta de rota RREP do AODV 25

Figura 12: O formato da mensagem de erro de rota RERR do AODV 25

Figura 13: Alterando o número de seqüência do destino 26

Figura 14: Alterando o número de saltos 27

Figura 15: Disposição dos sensores em um cluster 34

Figura 16: Disposição dos *clusters* de uma rede de sensores 34

Figura 17: O protocolo de distribuição de chaves 36

Figura 18: seqüência de mensagens para estabelecimento de rota e envio de dados 45

Figura 19: O Mica2 Mote da Crossbow Inc. 52

Figura 20: Disposição dos sensores num cluster modelo de simulação 53

Figura 21: Disposição dos *clusters* no plano 54

Figura 22: Disposição dos sensores para simulação 56

Gráfico 1: Tempo máximo para troca de uma mensagem CBR de 64 bytes sem rota pré-estabelecida 61

Gráfico 2: Tempo máximo para troca de uma mensagem CBR de 64 bytes com rota estabelecida 62

Gráfico 3: Tempo máximo para troca de uma mensagem CBR de 128 bytes sem rota pré-estabelecida 63

Gráfico 4: Tempo máximo para troca de uma mensagem CBR de 128 bytes com rota estabelecida 63

Gráfico 5: Tempo máximo para troca de uma mensagem CBR de 256 bytes sem rota pré-estabelecida 64

Gráfico 6: Tempo máximo para troca de uma mensagem CBR de 256 bytes com rota estabelecida 64

Gráfico 7: Tempo máximo para troca de uma mensagem CBR de 512 bytes sem rota pré-estabelecida 65

Gráfico 8: Tempo máximo para troca de uma mensagem CBR de 512 bytes com rota estabelecida 65

Gráfico 9: Tempo máximo para troca de uma mensagem CBR de 1024 bytes sem rota pré-estabelecida 66

Gráfico 10: Tempo máximo para troca de uma mensagem CBR de 1024 bytes com rota estabelecida 66

Gráfico 11: Envio de mensagem utilizando *cluster heads* de 8Mhz sem rota pré-estabelecida 67

Gráfico 12: Envio de mensagem utilizando *cluster heads* de 8Mhz com rota pré-estabelecida 68

Gráfico 13: Envio de mensagem utilizando *cluster heads* de 16 Mhz sem rota pré-estabelecida 68

Gráfico 14 : Envio de mensagem utilizando *cluster heads* de 16 Mhz com rota pré-estabelecida 69

Gráfico 15: Envio de mensagem utilizando *cluster heads* de 32 Mhz sem rota pré-estabelecida 69

Gráfico 16: Envio de mensagem utilizando *cluster heads* de 32 Mhz com rota pré-estabelecida 70

Gráfico 17: Envio de mensagem utilizando *cluster heads* de 64 Mhz sem rota pré-estabelecida 70

Gráfico 18: Envio de mensagem utilizando *cluster heads* de 64 Mhz com rota pré-estabelecida 71

LISTA DE TABELAS

Tabela 1: Comparação do tamanho de chaves em bits – RSA x ECC..... 15

Tabela 2: Valores de configuração dos sensores no NS-2 58

LISTA DE ABREVIAÇÕES

AC	Autoridade Certificadora
AODV	Ad hoc On Demand Distance Vector
AODV-EC	AODV Extra Cluster
AODV-IC	AODV Intra Cluster
CBR	Constant Bit Rate
CH	Cluster Head
DH	Diffie-Hellman
DoS	Deny of Service
ECC	Elliptic Curve Cryptography
EEPROM	Electrically-Erasable Programmable Read-Only Memory
ICP	Infraestrutura de Chaves Públicas
IP	Internet Protocol
KU	Chave Pública
KR	Chave Privada
LL	Link Layer
MD5	Message Digest 5
NS-2	Network Simulator 2
OTcl	Object Tool Command Language
PKI	Public Key Infrastructure
QoS	Quality of Service
RAM	Random Access Memory
ROM	Read-Only Memory
RFC	Request For Comments
RSA	Rivest Shamir Adleman
RERR	Route Error

RREP	Route Response
RREP-EC	Route Response Extra Cluster
RREP-IC	Route Response Intra Cluster
RREQ	Route Request
RREQ-EC	Route Request Extra Cluster
RREQ-IC	Route Request Intra Cluster
RREQ-INI	Route Request Initiator
SAODV	Secure AODV
SHA	Secure Hash Algorithm
TP	Texto Plano
UDP	User Datagram Protocol

RESUMO

Uma rede de sensores sem fio é uma coleção de dispositivos limitados em poder de processamento e alcance de transmissão, com baixa disponibilidade de memória e de energia. Devido a estas limitações, a maioria das soluções de segurança de redes cabeadas, tais como baseadas em ICP pura, não se aplicam diretamente neste tipo de ambiente. Este trabalho apresenta um protocolo híbrido que trata do esquema de gerenciamento de chaves e da interconexão transparente de *clusters*. Também é tratado o problema de captura de nós, oferecendo uma solução para a proteção da chave de grupo. Simulações mostram que aumentando o número de sensores na rede, o desempenho da comunicação entre dois *clusters* quaisquer permanece o mesmo.

ABSTRACT

A wireless sensor network is a collection of devices limited in low-powered batteries, processing, communication bandwidth capabilities and low memory availability. Due to these constraints, most of security approaches used in wired networks, such as those based on pure PKI, cannot be applied directly in this environment. In this work, we present a hybrid protocol that treats the group key management scheme and the transparent cluster interconnection. It also treats the node-tampering problem, offering a simple solution to control the group key. The feasibility of this protocol was verified by simulation and we concluded that increasing the number of nodes in the network does not change the performance of the interconnection between any two clusters.

1. INTRODUÇÃO

As redes de sensores sem fio estão se tornando cada vez mais comuns, seja em projetos de pesquisa, aplicações comerciais ou em projetos de defesa civil e militar. Podem ser utilizadas tanto no monitoramento de uma linha de montagem de uma fábrica quanto no auxílio para a observação da vida selvagem.

Este tipo de rede se caracteriza por ser formada por uma coleção de dispositivos, geralmente sensores, que não utilizam nenhuma infraestrutura de rede, ou seja, a comunicação é feita via ondas de rádio. Devido às características limitantes dos sensores, tais como o baixo poder de processamento, a baixa disponibilidade de memória e o consumo de energia baseado em baterias, as tentativas de apresentar uma solução para a interconexão destes dispositivos devem sempre levá-las em consideração.

Como os sensores possuem também uma limitação no alcance da transmissão, para que dois nós A e B, não adjacentes, possam se comunicar, se faz necessária a adoção de políticas de múltiplos saltos através dos nós intermediários entre eles. Neste sentido, os nós atuam não só como clientes e servidores mas também como roteadores, recebendo e repassando mensagens destinadas a outros nós.

Muitas aplicações de redes de sensores sem fio que têm seu foco no uso militar ou em ambientes hostis devem estar preparadas para lidar com a possibilidade de sofrer uma grande variedade de ataques maliciosos. Dentro desta característica, é imprescindível que a comunicação entre os nós seja feita de forma segura, onde alguns requisitos tais como confidencialidade, autenticidade, integridade e não-repúdio devem ser atendidos (Zhou e Haas, 1999). Confidencialidade significa que somente o nó a quem a mensagem está sendo destinada pode ter acesso a informação. Autenticidade diz respeito em poder identificar quem enviou a mensagem. Integridade é a garantia de que a mensagem não tenha sido modificada durante o trajeto. Não repúdio é a característica em que o emissor não pode negar que determinada mensagem tenha sido enviada por ele.

Para garantir a implementação desses requisitos de segurança, a tecnologia usada é a criptografia. Os dois ramos mais utilizados são a criptografia de chaves simétricas e a de chaves assimétricas ou públicas.

O uso de chaves públicas e suas derivações, utilizadas juntamente com funções *hash*, garantem as quatro características desejadas na comunicação segura. A garantia de integridade se obtém com o uso de funções *hash*, confidencialidade com criptografia e

autenticidade com assinatura digital. O não repúdio é garantido com o uso de criptografia e assinatura digital. Porém, uma vez que os nós de uma rede de sensores possuem limitações de processamento, de memória e de comunicação, a utilização de uma infraestrutura de chaves públicas (ICP) pura torna-se inviável pelo tempo de processamento e conseqüentemente pelo consumo de energia despendida para cifrar e decifrar as mensagens.

Uma maneira de reduzir estes gastos na troca de mensagens é a utilização de chaves simétricas, que devido a natureza das operações envolvidas tanto para cifrar quanto para decifrar, possuem um desempenho melhor.

Além disso, uma rede de sensores pode ser dividida em *clusters*, na tentativa de agrupar localmente os sensores pelas características dos serviços utilizados e providos, a fim de melhorar o desempenho (Bechler et al., 2004). Neste tipo de abordagem, é bastante útil a adoção de um dispositivo com menos restrições de processamento, memória e abrangência de transmissão do que os demais sensores do *cluster*. Este nó é chamado de *cluster head* (CH) e pode-se usar estas vantagens para torná-lo responsável pelo esquema de gerenciamento de chaves para os demais nós do *cluster*.

Devido à inviabilidade do uso de criptografia assimétrica em dispositivos limitados, são utilizadas chaves simétricas compartilhadas por todos os membros de um determinado *cluster* para a troca de mensagens. Com este tipo de abordagem, os quatro requisitos de segurança citados anteriormente também são assegurados, porém indiretamente. O uso de funções *hash* garante integridade das mensagens. Para os demais requisitos, são consideradas as relações de confiança entre os nós, ou seja, nenhum nó anômalo possui uma chave de grupo. Esta confiabilidade deve ser garantida na hora do estabelecimento da chave simétrica (Hu e Sharma, 2005). O uso de chaves de grupo simétricas acarreta dois principais problemas. Um diz respeito a captura de nós, uma vez obtida a chave de grupo, todo *cluster* fica comprometido. O outro problema é a formação de grupos fechados de comunicação, sendo que somente quem possui a chave de grupo pode se comunicar.

1.1. OBJETIVOS

O objetivo principal deste trabalho é apresentar uma solução que garanta a comunicação segura numa rede de sensores sem fio dividida em *clusters*, bem como obter e avaliar informações de desempenho relativas à interconexão destes *clusters* de forma transparente.

Como objetivos específicos deste trabalho, podem ser citados os procedimentos necessários para o desenvolvimento desta solução. Dentre eles pode-se destacar:

- definir o conjunto de mensagens e desenvolver um protocolo híbrido para a distribuição e gerenciamento das chaves de grupo para os sensores do *cluster*;
- definir o conjunto de mensagens e desenvolver um protocolo híbrido para o estabelecimento de rotas entre os sensores comunicantes, tomando por base o protocolo AODV e as extensões seguras do SAODV, inclusive fazer as alterações necessárias para que o protocolo proposto possa funcionar no ambiente de *clusters* que está sendo proposto;
- escolher e aprender uma ferramenta de simulação adequada para o ambiente que se quer simular;
- definir a disposição dos sensores no *cluster*, bem como do *cluster head*;
- definir as variações do número de *clusters* que serão simulados;
- obter resultados das simulações e confrontá-los entre si para que possam ser ilustrados graficamente;
- contribuir para a evolução e amadurecimento dos temas tratados neste trabalho, tanto em nível teórico quanto experimental.

1.2. METODOLOGIA

A realização deste trabalho pode ser dividida em etapas bem definidas, seguindo a metodologia ilustrada no diagrama 1. A primeira etapa deste trabalho foi a delimitação do escopo da pesquisa. Como há uma enorme variedade de trabalhos e abordagens relacionados a redes ad hoc e de sensores, tais como, qualidade de serviços, segurança, alta disponibilidade, dentre outras, foi decidido trabalhar na área de segurança aplicada a uma rede de sensores sem fio.

Após ter sido delimitado o escopo da pesquisa, houve a necessidade de encontrar os trabalhos já desenvolvidos nesta área e conseqüentemente, conhecer o que estava sendo feito na área de segurança voltada para redes de sensores sem fio. Este levantamento bibliográfico usou de diversos periódicos impressos, repositórios de documentos disponíveis na Internet e artigos encontrados nas páginas das próprias universidades dos seus autores, bem como nos

anais dos eventos mais importantes na área de segurança e de gerência de redes. Estes periódicos e eventos importantes são apresentados mais adiante, no ANEXO IV deste trabalho.

De posse dos artigos que tratam de segurança em redes de sensores e redes ad hoc, foram identificados os trabalhos relacionados com o tema da pesquisa. Esta atividade inclusive ajudou muito a definir o foco da pesquisa, no caso o gerenciamento de chaves e roteamento seguro.

As duas etapas seguintes foram executadas de maneira simultânea. Como já tinha sido definido que as proposições seriam avaliadas via simulações, houve a necessidade de estudar a ferramenta *Network Simulator 2* (NS-2), para já ir vislumbrando como poderiam ser feitas as simulações, bem como desenvolver um conhecimento suficiente para que os ambientes que se desejassem simular fossem realmente transportados no simulador. Paralelamente ao estudo do NS-2, foi sendo elaborada e desenvolvida a solução propriamente dita. Esta fase será melhor descrita adiante, tendo por base o diagrama 2.

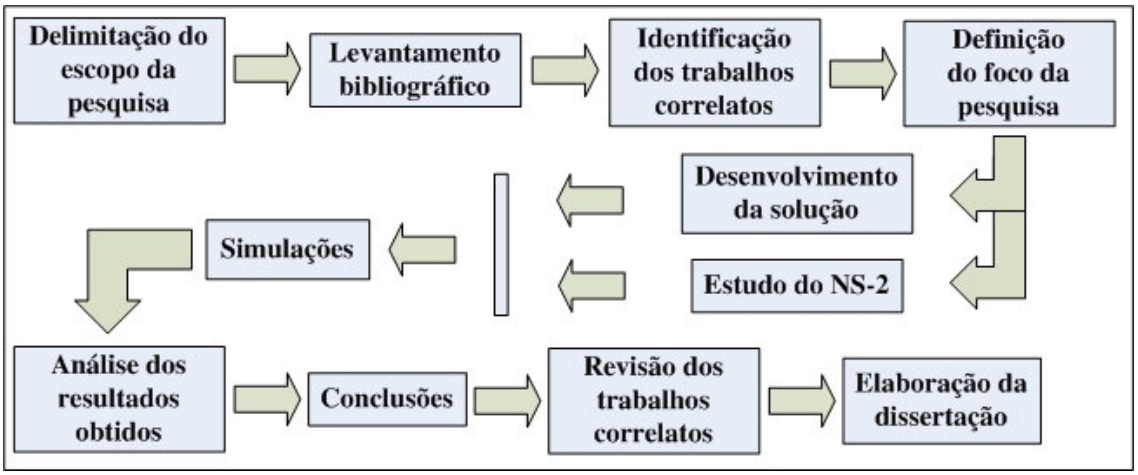

Diagrama 1: Trajetória da pesquisa

A parte da metodologia que se baseia nas simulações também será abordada posteriormente neste capítulo, utilizando a ilustração do diagrama 3 para melhor exemplificar como as mesmas foram feitas. Tomando por base os resultados das simulações, e transportando-os para uma ferramenta de planilha eletrônica, os mesmos puderam ser melhor analisados e confrontados entre si. Foi possível tirar as conclusões específicas sobre o desempenho desta solução na parte de interconexão dos *clusters*. O foco do trabalho foi o

estabelecimento seguro de rotas, e a interconexão dos sensores, sendo que o foco das simulações foi a interconexão segura dos sensores estando em *clusters* distintos.

Como última etapa deste foi a elaboração do texto escrito da dissertação. Antes de ser totalmente finalizada, houve ainda uma preocupação em pesquisar trabalhos correlatos, que pudessem ter sido desenvolvidos neste intervalo de tempo desde a primeira etapa deste trabalho.

Na etapa de desenvolvimento da solução, ilustrada no diagrama 2, primeiro se buscou um esquema para a comunicação segura entre os sensores divididos em diversos *clusters*. Tendo por base o protocolo de comunicação AODV, que é aplicado a redes ad hoc e não necessariamente a redes de sensores para o estabelecimento de rotas. Foi estudada também a proposta abordada nas extensões do SAODV, que se baseia numa solução de chaves públicas para garantir a segurança do protocolo AODV. Como os próprios autores enfatizam que esta não é a melhor solução para dispositivos limitados como os sensores, uma solução utilizando chaves públicas e chaves secretas foi sendo melhor projetada.

Levando em conta a topologia adotada para cada *cluster*, onde há um dispositivo com menos restrições que os demais sensores deste cluster, foi projetada uma solução híbrida. Esta solução utilizava por base a proposição do SAODV para a comunicação entre os *clusters* (*extra-cluster*) e uma modificação do SAODV, baseada em chaves simétricas, para a comunicação dentro de cada *cluster* (*intra-cluster*). Desta forma, houve a necessidade de desenvolver um protocolo que disponibilizaria uma seqüência de mensagens que deveriam ser trocadas entre os sensores para estabelecimento destas chaves simétricas.

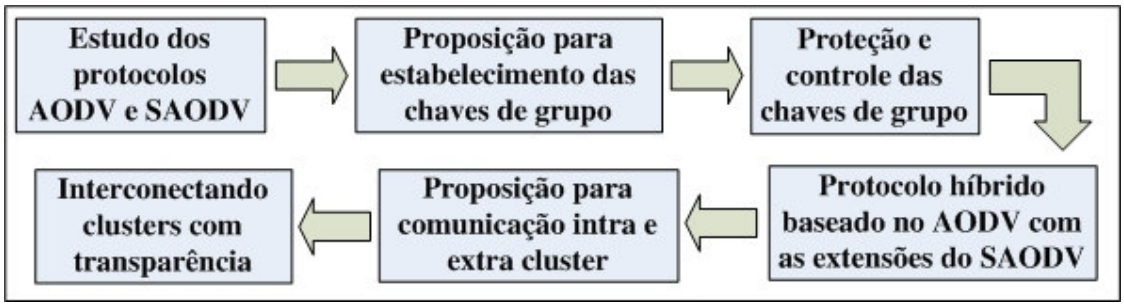

Diagrama 2: Desenvolvimento da solução

Depois de definida como se daria a obtenção das chaves de grupo (simétricas), houve a necessidade de propor uma maneira de garantir a proteção contra a distribuição inadvertida de mais chaves de grupo por parte do *cluster head*. Esta solução também pôde ser utilizada

para ajudar no controle da redistribuição das chaves de grupo (*rekeying*), que pode ser feita de tempos em tempos.

Quando o foco do desenvolvimento se voltou para a interconexão dos *clusters*, houve a necessidade de um aperfeiçoamento do protocolo *extra-cluster* para que o mesmo pudesse funcionar corretamente. Este fator foi necessário principalmente pelo alcance de transmissão do *cluster head*. Superado este problema com o protocolo de roteamento e aprimorado o mesmo, deu-se o início das simulações.

Uma das partes da metodologia utilizada neste trabalho é baseada na coleta de informações obtidas através de simulações feitas utilizando o NS-2. Os passos envolvidos nesta metodologia relacionados com as simulações estão ilustrados no diagrama 3. O primeiro passo foi definir o que deve ser simulado. Os ambientes montados para as simulações dizem respeito à interconexão dos *clusters*.

Uma vez definido o que se quer simular, se deve definir a disposição dos sensores no *cluster*, priorizando o espalhamento uniforme e eqüidistante dos sensores. As simulações se baseiam na interconexão de dois até cem *clusters*, variando também a capacidade de processamento dos dispositivos menos restritos, responsáveis pelo estabelecimento de rotas entre os *clusters* e o tamanho dos pacotes de dados transmitidos. Esta etapa de configuração dos sensores e *cluster heads* leva em conta o posicionamento físico e também as características dos dispositivos com relação a capacidade e abrangência de transmissão.

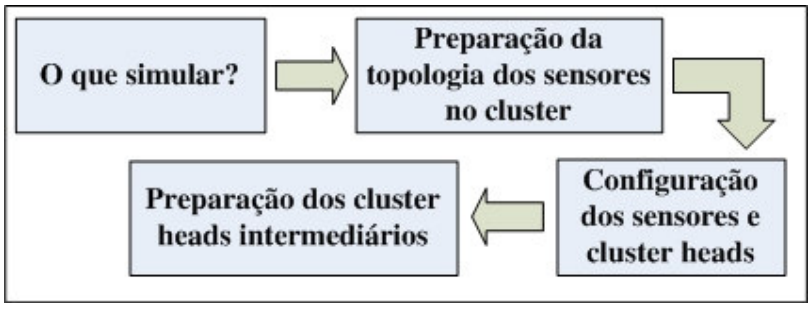

Diagrama 3: As simulações

Para a implementação do protocolo, é utilizada a versão do protocolo AODV que já vem implementada no NS-2. Por ser um simulador discreto, que conta com um mecanismo de medição do tempo de simulação baseado nos atrasos de cada operação e da propagação das mensagens através do espaço, torna-se necessário encontrar na literatura valores relativos ao desempenho das diversas funções criptográficas quando aplicada aos sensores.

As demais etapas posteriores às simulações, a saber, análise dos resultados obtidos, conclusões e elaboração da dissertação já foram mencionadas acima e serão melhor apresentadas nos capítulos posteriores desta dissertação.

1.3. JUSTIFICATIVA E RESULTADOS ESPERADOS

Este trabalho se justifica pelo fato de ainda não ter sido encontrada uma solução para o roteamento entre *clusters* numa rede de sensores feito pelo *cluster head* considerando a possibilidade de mobilidade dos sensores. Também ainda não foi encontrado um conjunto de resultados demonstrando o desempenho da interconexão segura de vários *clusters*. Algumas soluções encontradas na literatura apontam para a comunicação dos *clusters* feita pelos sensores que estão situados nas periferias do seu *cluster* e conseqüentemente mais próximos dos *clusters* vizinhos.

Neste trabalho os *cluster heads* são considerados como entidades responsáveis pela interconexão segura entre os *clusters*, abrangendo desde o estabelecimento das chaves de grupo, passando pela descoberta e estabelecimento de rotas de comunicação, até a troca de informações seguras entre os sensores comunicantes estando estes em *clusters* diferentes.

Com esta abordagem, espera-se que o protocolo proposto para o estabelecimento das chaves de sessão realmente satisfaça todas as condições que possam acontecer na inclusão de novos sensores. Também espera-se que a realocação de sensores em outros *clusters* possa ser satisfeita através do protocolo proposto.

Com relação ao roteamento, se espera que o protocolo de roteamento proposto, baseado no protocolo AODV e nas extensões seguras do SAODV, possa funcionar adequadamente nas comunicações internas em cada *cluster* utilizando chaves simétricas. O mesmo espera-se para as comunicações *extra-clusters*, só que utilizando as mensagens baseadas em chaves públicas para estabelecimento de rotas. E o mais importante, espera-se que esta conversão de protocolo seja feita pelos *cluster heads* de maneira transparente.

Um dos principais atrativos desta solução e o resultado mais esperado dizem respeito ao funcionamento do protocolo de roteamento. Espera-se que o estabelecimento de uma rota entre dois sensores quaisquer, situados em *clusters* diferentes, tenha um tempo total quase constante, sem que entre estes dois sensores exista alguma rota pré-estabelecida. Isto se deve à característica da solução proposta, onde os *cluster heads* dos *clusters* que querem se comunicar (origem e destino) é que são responsáveis pela execução das funções criptográficas

mais dispendiosas em tempo e energia, no caso aquelas baseadas na tecnologia de chaves públicas.

No caso do repasse das mensagens feitas pelos *cluster heads* intermediários, quando houver necessidade, as funções que são executadas por estes *cluster heads* intermediários são funções *hash*, sabidamente na ordem de milhares de vezes mais rápidas que as baseadas em chaves públicas. Desta forma, não importa quantos *clusters* estarão entre os dois *clusters* que querem se comunicar. O tempo despendido para toda a descoberta de uma rota estará concentrado nas operações executadas nos *clusters* de origem e de destino. Com isso, as duas únicas variáveis que alteram os tempos totais na interconexão de diversos *clusters* são o tempo de propagação das mensagens no espaço e a aplicação das funções *hash* pelos *cluster heads* intermediários.

1.4. LIMITAÇÕES

As principais limitações desta solução que está sendo apresentada dizem respeito a utilização de um único elemento responsável pelo gerenciamento das chaves e pela interconexão de cada *cluster*. Possivelmente o *cluster head* de cada *cluster* poderá se tornar um gargalo na comunicação, pois toda a interconexão entre *clusters* deve ser feita pelos *cluster heads*. Também não está sendo considerada a possibilidade de captura de um *cluster head*.

Outra limitação que pode ser destacada é a maneira como são tratadas as autenticações de mensagens entre os *cluster heads* intermediários, na comunicação *extra-cluster*, podendo haver um retardo na identificação de uma mensagem anômala. Isto se deve ao fato de se procurar a maior eficiência do protocolo. Ou seja, primeiro se repassa a mensagem, para depois verificar a integridade e autenticidade da mesma. Porém, este retardo na averiguação da autenticidade das mensagens não provoca construções de tabelas de rotas falsas.

1.5. ESTRUTURA DA DISSERTAÇÃO

Este trabalho está organizado em capítulos numerados de 1 a 6, sendo que o primeiro capítulo trata de introduzir o tema que será abordado. São expostos na seqüência o objetivo principal e os objetivos específicos, seguidos da metodologia de pesquisa aplicada. Em seguida a justificativa e as expectativas e os resultados esperados são apresentados. Seguindo

a seqüência do capítulo primeiro, os fatores limitantes do trabalho são tratados, expondo os eventuais temas não abordados neste trabalho.

No capítulo segundo é feita uma revisão bibliográfica que trata dos conceitos e definições na área de segurança e criptografia, cruciais para o entendimento do trabalho.

No terceiro capítulo, são apresentadas as tecnologias de roteamento em redes sem fio e ainda apresenta os trabalhos correlatos aos temas que estão sendo tratados neste trabalho.

O quarto capítulo é dedicado exclusivamente ao desenvolvimento da solução proposta. Esta solução abrange o modelo de distribuição dos sensores no *cluster*, a disposição dos *clusters* na rede e o modelo de sensor que foi a base das pesquisas feitas. A solução ainda abrange o esquema de distribuição de chaves e a interconexão de *clusters*.

No quinto capítulo são apresentadas as simulações feitas e os resultados obtidos. É apresentada também a ferramenta utilizada para as simulações e o ambiente simulado.

Este trabalho se encerra com as conclusões feitas no capítulo 6, baseadas nos resultados obtidos pelas simulações .

2. SEGURANÇA EM AMBIENTES DISTRIBUÍDOS

O capítulo de revisão bibliográfica pode ser dividido em três partes principais. A primeira, diz respeito aos assuntos e definições relevantes e que ajudam a melhor compreensão do que está sendo tratado neste trabalho. A segunda parte traz os periódicos e eventos importantes, relacionados com o tema deste trabalho. Já na terceira parte deste capítulo, são apresentados os trabalhos relacionados com o tema, divididos de acordo com as abordagens que serviram de base para o desenvolvimento deste trabalho.

2.1. DEFINIÇÕES IMPORTANTES

2.1.1. Requisitos de Serviços Seguros e Ataques de Segurança

De acordo com Stallings (1999), os requisitos para os serviços serem considerados seguros, podem ser classificados em: confidencialidade, autenticação, integridade, não-repúdio, controle de acesso e disponibilidade.

Assegurando o requisito de confidencialidade, garante-se que a informação é acessível (leitura) somente para as partes autorizadas. Com autenticação, a origem de uma mensagem ou documento eletrônico é corretamente identificada, com a garantia de que a identidade não é falsa. Através da integridade, somente as partes autorizadas são habilitadas a modificar (escrita) uma informação, não sendo possível sofrer uma modificação no meio do percurso sem que seja detectada. O não repúdio diz respeito a que tanto quem envia quanto quem recebe uma mensagem, não pode negar que houve a transmissão. O controle de acesso garante que à informação, o acesso seja controlado por ou para um sistema alvo. Já a disponibilidade requer que uma informação ou um sistema esteja disponível para uma parte autorizada quando necessário.

Ainda usando Stallings (1999) como referência, os ataques de segurança podem ser classificados em quatro tipos diferentes. A figura – A, está representado o fluxo normal da comunicação entre uma entidade origem e outra destino. A figura 1 – B, traz um ataque de interrupção, onde um recurso do sistema ou é destruído, tornando-se indisponível ou é inutilizável. Este tipo de ataque interfere no requisito de disponibilidade de sistemas seguros. Um ataque de interceptação é ilustrado na figura 1 – C, onde uma entidade não autorizada ganha acesso a um recurso. Este é um ataque contra o requisito de confidencialidade. Uma parte não autorizada pode ser uma pessoa, um programa ou um computador. Na figura 1 – D,

está ilustrado um ataque de modificação. Nele, uma parte não autorizada não somente obtêm acesso a um recurso como o altera. Este tipo de ataque infringe o requisito de integridade. O último tipo de ataque que está ilustrado na letra E, o de fabricação, mostra que uma parte não autorizada insere informações falsas no sistema. Este ataque age contra o requisito de segurança de autenticidade.

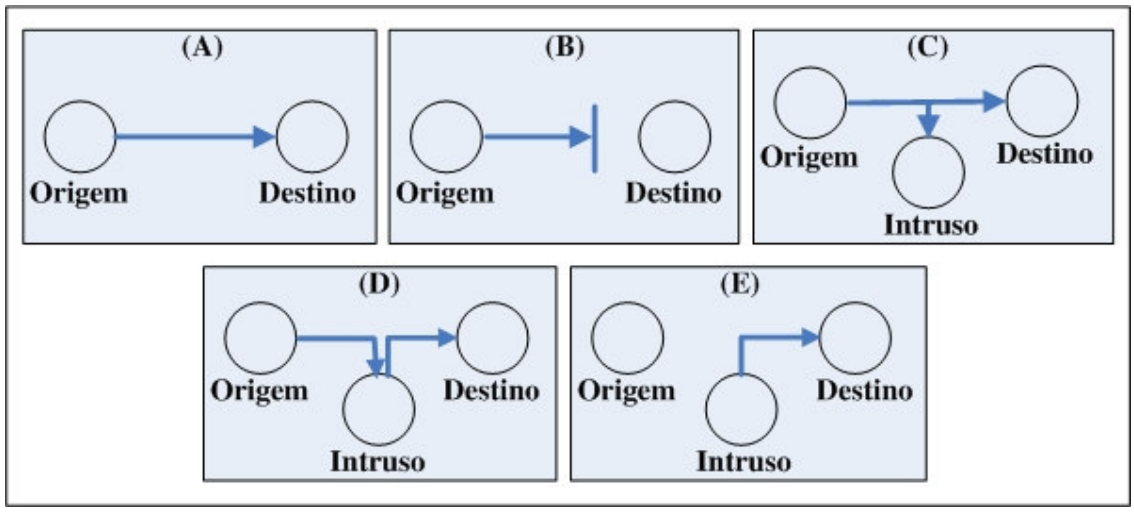

Figura 1: Os quatro tipos de ataques de segurança

Usando como referência os tipos de ataque ilustrados na figura 1, os ataques ainda podem ser classificados de outra maneira: ataques passivos e ativos. Os ataques passivos têm como objetivo a obtenção da informação que está sendo trocada, sem alteração dos mesmos. Esse tipo de ataque diz respeito a tudo que envolva bisbilhotagem e monitoramento de transmissões.

Desde a liberação de conteúdos de mensagens, tais como uma conversa telefônica, um e-mail transmitido que contenham informações confidenciais, até a análise de tráfego, que é mais sutil, onde o intruso captura as informações e as analisa por conta própria. Os ataques passivos são difíceis de detectar justamente porque não envolvem alteração dos dados que estão sendo transmitidos. Para combater estes ataques, é preferível a prevenção do que a detecção.

Já os ataques ativos são aqueles que envolvem algum tipo de modificação das informações que estão sendo trocadas ou criação de novas informações. Este tipo de ataque pode ser dividido em quatro categorias: mascaramento, repetição, modificação de mensagens e DoS.

Um mascaramento ocorre quando uma entidade se faz passar por outra entidade diferente. Geralmente ocorre envolvendo outras categorias de ataques ativos. O ataque de repetição envolve a captura passiva de informações e sua retransmissão subseqüente para produzir um efeito não autorizado. A modificação de mensagens é um tipo de ataque onde alguma parte da informação que está sendo trocada é capturada e alterada, para que possa produzir um efeito não autorizado. Nos ataques de DoS o intruso tem um alvo específico. Pode ser um destino particular, uma subrede, etc. As entidades alvo deste tipo de ataque são atingidas através da desabilitação da rede que estão conectadas ou através do sobrecarregamento através de mensagens a fim de degradar o desempenho.

Ataques ativos apresentam características opostas dos ataques passivos. Enquanto ataques passivos são difíceis de detectar, há medidas que previnem seu sucesso. Por outro lado, é bastante difícil prevenir ataques ativos, porque requerem proteção física de todos os meios de comunicação. O objetivo é detectar e reparar qualquer alteração ou repasse causado por eles.

2.1.2. Criptografia

Criptografia é a tecnologia composta por um conjunto de técnicas aplicadas a uma informação com a intenção de mascarar e ocultar esta informação de entidades que não devem ter acesso a ela.

De acordo com Menezes et al. (1997), a criptografia é o estudo de técnicas matemáticas relacionadas a aspectos de segurança da informação tais como confidencialidade, integridade de dados, autenticação de entidades e da origem dos dados.

Os sistemas de criptografia podem ser classificados de diversas maneiras diferentes e independentes. A primeira classificação que pode ser feita diz respeito ao tipo de operações utilizadas para transformar um texto plano num texto cifrado. Os algoritmos de criptografia são baseados em dois princípios gerais, substituição e transposição. Substituição ocorre quando um elemento do texto plano é mapeado em outro elemento. Transposição é feita através do rearranjo dos elementos de um texto plano.

Os algoritmos utilizam combinações de substituição e transposição, aplicadas múltiplas vezes sobre o mesmo texto ou parte dele. Outra forma de classifica-los é considerar como o texto plano é processado. Pode ser processado por blocos, no qual um bloco de texto plano processado pode servir como entrada para o processamento de outro bloco.

Uma outra possível classificação é caracterizada pelo número de chaves utilizadas. Nos casos em que tanto o emissor quando o receptor utilizam uma mesma chave para cifrar e decifrar, esta criptografia é dita simétrica, de chave única, chave secreta ou convencional. Caso utilizem duas chaves, diferentes e complementares, em que o texto que for cifrado por uma chave, somente a sua complementar pode decifrar e vice-versa, esta criptografia é dita assimétrica, de duas chaves ou de chaves públicas.

2.1.2.1. Criptografia simétrica

A criptografia simétrica é caracterizada pela adoção de um mesmo segredo ou chave para cifrar e decifrar um texto plano. Devido à natureza das operações efetuadas sobre o texto que se quer cifrar ou decifrar, este tipo de criptografia é muito eficiente.

Como a chave precisa ser compartilhada entre os dois participantes da comunicação, a distribuição e o gerenciamento desta chave se torna a tarefa mais difícil e desafiadora neste tipo de criptografia. A figura 2 ilustra este tipo de criptografia.

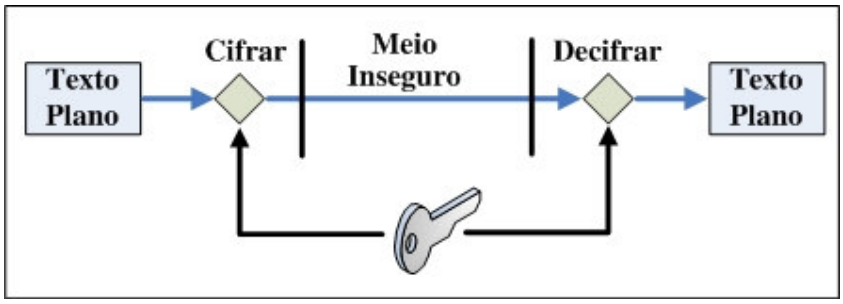

Figura 2: Criptografia com chave simétrica

2.1.2.2. Criptografia assimétrica

Também conhecida como criptografia de chaves públicas ou de duas chaves. Como o próprio nome sugere, utiliza-se de duas chaves para o processo de cifrar e decifrar as informações. Uma chave pública (KU), que todos podem ter acesso e outra privada (KR) que deve ser mantida em segredo. Tendo a chave privada, é possível gerar a chave pública. Mas não é possível obter uma chave privada conhecendo-se a chave pública.

A dificuldade está justamente no tempo necessário para determinar a chave privada a partir da chave pública. Quanto maior o tamanho da chave, mais difícil se torna a sua

descoberta. A principal característica deste tipo de criptografia é que todo texto plano cifrado com uma chave só pode ser decifrado pela outra correspondente do par.

Pode ser utilizada tanto para confidencialidade quanto para autenticação. Depende da maneira como as chaves são utilizadas. Para garantir confidencialidade (cifrar), sempre que uma entidade A (Alice) quiser cifrar um texto plano para enviar para outra entidade B (Bob), ela utiliza a chave pública de Bob para fazê-lo.

Bob utiliza sua chave privada para verificar a informação. Como somente Bob possui sua chave privada, ninguém mais tem acesso à informação. A figura 3 ilustra o uso de chaves assimétricas para garantir confidencialidade.

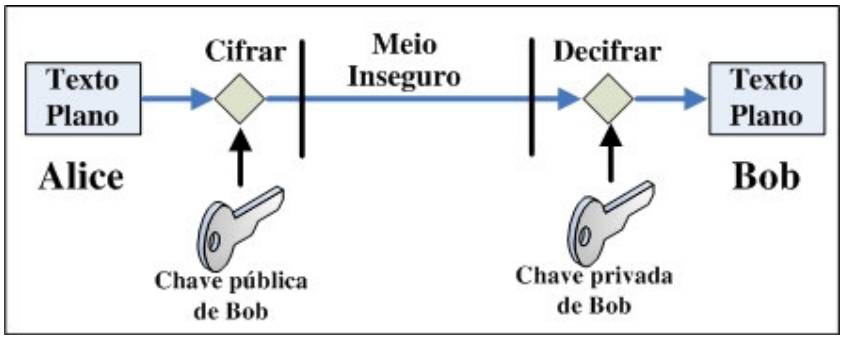

Figura 3: Uso de criptografia assimétrica para garantir confidencialidade

Quando Alice quiser mandar uma informação para Bob e fazer com que Bob tenha certeza que foi Alice quem mandou (autenticidade), ela utiliza a sua chave privada para cifrar a mensagem e Bob ao decifrar a mensagem utilizando a chave pública de Alice passa a ter certeza de quem enviou a mensagem. A figura 4 ilustra a garantia de autenticidade através do uso de chaves assimétricas.

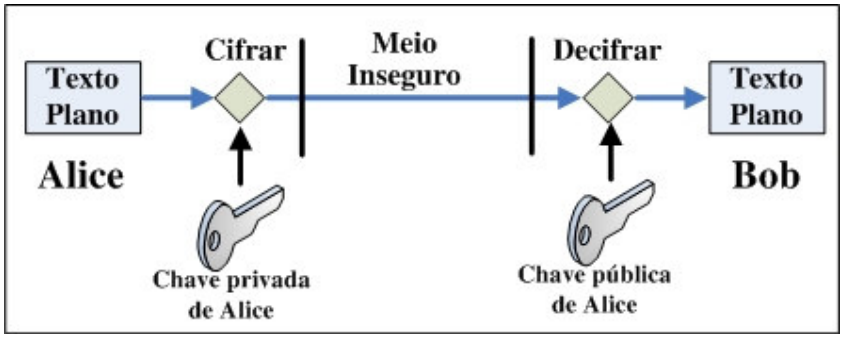

Figura 4: Uso de criptografia assimétrica para garantir autenticidade

- Dois dos algoritmos de chaves públicas mais conhecidos: o RSA e o ECC

O algoritmo criptográfico RSA é baseado em operações matemáticas exponenciais sobre número primos bastante grandes. Nesta tecnologia, dados KU={e,n} e KR={d,n}, a operação sobre blocos de k bits, com $2^k < n \leq {}^{2k+1}$, tem-se que para cifrar um texto plano (TP), o texto cifrado (C) seria obtido operando:

$$C = TP^e \bmod n$$

Dado o texto cifrado (C), para obter o texto plano opera-se:

$$TP = C^d \bmod n = (TP^e)^d \bmod n = TP^{ed} \bmod n$$

Com os valores acima apresentados, é possível encontrar e, d, n tal que:

$$TP^{ed} = TP \bmod n \text{ para todo } TP < n$$

Não é difícil calcular TP^e e C^d para todos os valores de TP < n, mas é improvável determinar d tendo-se os valores de e,n. Os números primos utilizados na obtenção de e,d precisam ser grandes para que a obtenção de d seja computacionalmente impossível.

O algoritmo criptográfico de curvas elípticas (ECC), é baseado em operações matemáticas de soma e multiplicação escalar. Dado um ponto P sobre a curva, a ele é multiplicado um valor k, obtendo outro ponto Q sobre a curva. Cada curva possui um ponto G, também chamado de ponto de base. A geração das chaves é feita pela obtenção de um número randômico k, que será a chave privada e a partir deste valor de k calcula-se kG, que é a chave pública. A segurança desta abordagem está na dificuldade para resolver o problema de logaritmos discretos em curvas elípticas, ou seja, encontrar k dados P e Q=kG. Este problema é computacionalmente intratável.

Na comparação dos dois métodos, os dois principais atrativos são o tamanho das chaves e as operações matemáticas necessárias para implementação dos algoritmos. A tabela 1 mostra os valores dos tamanhos de chaves para o RSA e ECC, em bits, ao qual cada coluna corresponde a segurança obtida [CERTICOM, 2006].

RSA	1024	2048	3072	8192	15360
ECC	160	224	256	384	512

Tabela 1: Comparação do tamanho de chaves em bits – RSA x ECC

Pode-se observar que a utilização de uma chave de 1024 bits no RSA alcança-se o mesmo nível de segurança que o oferecido pelo ECC quando utiliza uma chave de 160 bits. Da mesma forma, para uma chave de 512 bits no ECC, precisaria de uma chave de 15360 bits no RSA para equiparar o nível de segurança oferecido.

Devido ao fato das operações exponenciais do RSA serem mais dispendiosas do que as multiplicações escalares do ECC, esta tecnologia torna-se mais atraente quando consideramos sensores com poder de processamento limitado.

2.1.2.3. Funções resumo (*hash*)

No contexto de segurança, se alguma informação, após ter sido gerada, for modificada por qualquer entidade externa, é possível detectar que houve modificação através do uso das funções resumo.

As funções resumo são constituídas por operações criptográficas simples, mas que combinadas se tornam eficientes e seguras. Sempre produzem resultados diferentes para textos diferentes. Independentemente do tamanho do texto ao qual a função resumo está sendo aplicada, o resultado é produzido com tamanho fixo. A partir do *hash* gerado é impossível descobrir o texto plano que o originou. A figura 5 ilustra o uso de uma função *hash* sobre um texto plano de tamanho variável.

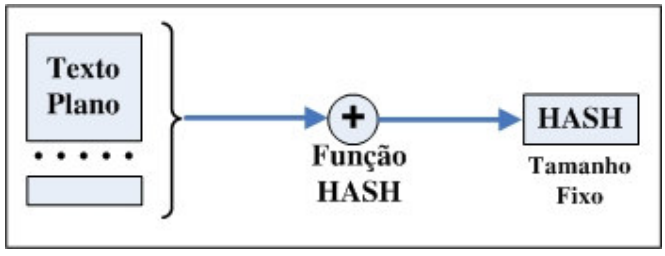

Figura 5: Uma função *hash* aplicada sobre um texto plano

As funções *hash* não podem ser utilizadas sozinhas, pois um intruso pode modificar os dados e gerar um novo *hash* sobre os dados modificados. Desta forma, será garantida a integridade dos dados alterados e não dos originais.

As funções *hash* podem utilizar chaves ou não. As duas funções *hash* mais conhecidas e utilizadas são MD5 e SHA, não utilizam chaves, portanto são de conhecimento público. Pela característica de não ser possível obter o texto original conhecendo-se apenas o seu *hash*,

a divulgação de qual função foi utilizada para gerar o *hash* não é um problema, ao contrário, facilita a verificação da integridade dos dados por parte do destinatário.

2.1.2.4. Assinatura digital

Assinatura digital é uma técnica criptográfica usada para garantir autenticidade, integridade e não repúdio. De acordo com Stalings (1999), a assinatura digital é o desenvolvimento mais importante do trabalho baseado em chaves públicas.

A assinatura digital combina a tecnologia de funções *hash* com a criptografia de chaves públicas. Qualquer entidade pode facilmente verificar a autenticidade de uma assinatura digital, enquanto a sua falsificação por uma entidade intrusa é computacionalmente impossível.

As operações criptográficas envolvidas na assinatura e na verificação de uma assinatura são ilustradas na figura 6. Primeiro, a entidade emissora (Alice) calcula o *hash* aplicado sobre um texto plano que deseja ser transmitido. Este *hash* é cifrado, usando a chave privada do emissor. A assinatura digital é concatenada ao texto plano e a mensagem é enviada.

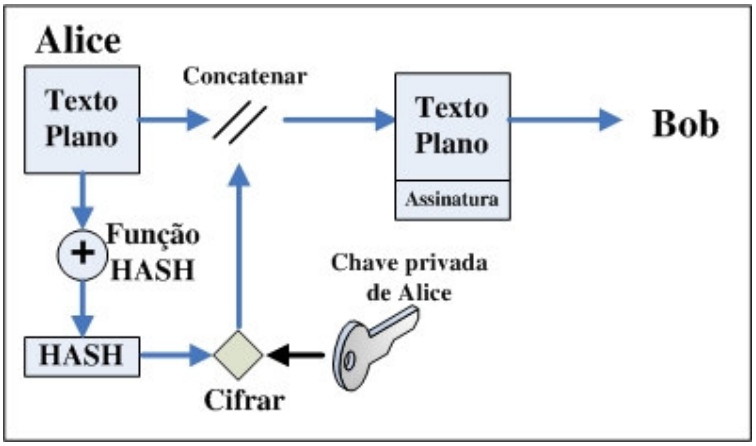

Figura 6: Criação de uma assinatura digital

Quando a entidade receptora (Bob) recebe a mensagem, ela calcula o *hash* do texto plano e ao mesmo tempo decifra a assinatura usando a chave pública do emissor. Então compara os dois resultados obtidos. Se o *hash* gerado através do texto plano for igual ao *hash* obtido com a assinatura que foi decifrada, tanto autenticidade quanto integridade foram atendidos. A verificação da assinatura digital é ilustrada na figura 7.

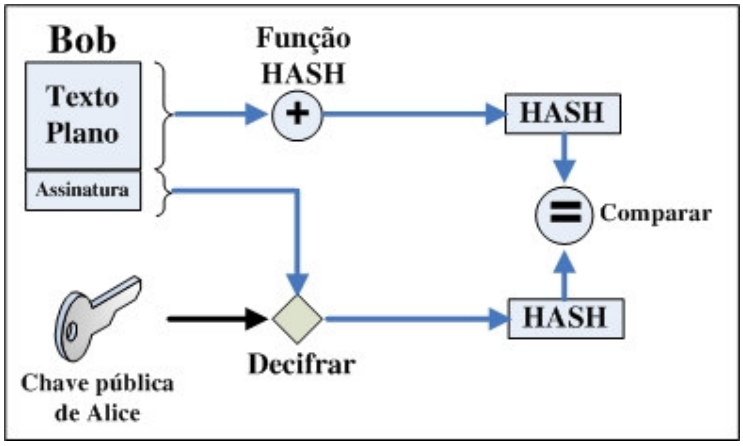

Figura 7: Verificação de uma assinatura digital

2.1.2.5. Troca de Chaves de Diffie-Hellman

Em 1976, Whitfield Diffie e Martin Hellman publicaram seu primeiro trabalho que apresenta um modelo de troca de chaves sobre um canal desprotegido de comunicação. A segurança do esquema de Diffie-Hellman é baseado na dificuldade de computação de logaritmos discretos.

Considerando Alice e Bob, dois nós que desejam estabelecer uma chave sobre um meio inseguro. Primeiro, Alice e Bob acordam sobre o uso de um número primo q e uma base g. Cada um, gera um número randômico X, menor que q. Este valor é a chave privada de cada entidade(X_A de Alice e X_B de Bob).

A partir da chave privada, calcula-se a chave pública Y. Sendo Y a chave pública, esta pode ser divulgada para qualquer entidade. Portanto, Bob possui a chave pública de Alice(Y_A) e Alice possui a de Bob(Y_B). Cada entidade pode calcular uma chave idêntica K. Bob obtém $K = (Y_A)^{X_B} \bmod q$. Da mesma forma, Alice obtém $K = (Y_B)^{X_A} \bmod q$.

Desta forma, como cada entidade utiliza sua chave privada como base de cálculo para obter a chave compartilhada K, e como a chave privada só é conhecida pela entidade que a detém, somente as duas partes comunicantes podem obter a mesma chave secreta K.

2.1.2.6. Infraestrutura de chaves públicas - ICP

O uso de criptografia assimétrica e suas derivações traz um problema que deve ser tratado: a autenticidade das chaves públicas. Deve haver um mecanismo para garantir que a

entidade que detém a chave pública é realmente quem diz ser. Para isso há a necessidade de uma infraestrutura para que a autenticidade das chaves públicas seja garantida.

Uma ICP baseia-se numa estrutura hierárquica de confiança entre as entidades que a compõem. Duas entidades comunicantes, para que tenham certeza da identidade uma da outra, precisam ter suas identidades confirmadas por uma terceira entidade que seja de confiança de ambas. Esta terceira entidade é conhecida como Autoridade Certificadora (AC).

Uma AC emite um certificado digital para uma entidade a fim de identifica-la e autenticá-la. Para que seja possível verificar a autenticidade de um certificado digital, é preciso que ambas as entidades tenha uma AC de confiança em comum às duas entidades comunicantes.

Numa ICP, há uma Autoridade Raiz, e a partir dela todas as outras AC são derivadas, ou seja autenticadas por ela ou por alguma que tenha sido autenticada por ela. Neste caso, há um caminho de certificação para qualquer AC válida. Quando for necessário validar um certificado digital, percorre-se o caminho de certificação no sentido contrário ao que ele foi criado, até que se encontre uma AC em comum.

- Certificado Digital

É um documento com formato padronizado (X-509), legível, assinado por uma AC, que serve para identificar uma entidade. Sua principal função é autenticar a chave pública da entidade ao qual ele foi emitido, bem como identifica-la. Contém uma série de campos, entre eles: nome, chave pública, nome da entidade certificadora, assinatura da entidade certificadora, período de validade, número de série, nome do domínio e algoritmo utilizado.

No estabelecimento de uma comunicação segura, as entidades envolvidas precisam conhecer a chave pública uma da outra. Para isso, cada entidade comunicante recebe o certificado digital da outra entidade. Após recebê-lo, é preciso verificar a autenticidade do certificado, isto é, ter certeza que o certificado tenha sido emitido e assinado por uma AC de confiança da entidade que o está verificando.

2.1.3. Gerenciamento de chaves

O gerenciamento de chaves é um tema amplo e complexo. A idéia principal é que os sistemas que tentam implementar segurança através de criptografia, precisam ter essas chaves distribuídas entre os nós da rede. De acordo com Menezes, Van Oorschot e Vanstone (1997),

o termo gerenciamento de chaves está relacionado com: a inicialização dos usuários dos sistemas num domínio; geração, distribuição e instalação das chaves; controle do uso das chaves; atualização, revogação e destruição das chaves; e armazenamento, backup/restauração e arquivamento das chaves.

Dependendo do ambiente, o enfoque dado para o gerenciamento de chaves muda completamente. Como o trabalho desenvolvido é na área de redes ad hoc e de sensores, o gerenciamento de chaves pode ser dividido em dois grupos principais. No primeiro, as chaves pré-distribuídas abrangem a rede inteira e no outro grupo as chaves são distribuídas para cada nó específico.

No primeiro grupo, em que cada nó na rede possui a mesma chave criptográfica, o problema está na revelação desta chave que acaba por comprometer todo o segredo da rede. O segundo grupo supõe uma chave específica para cada par de nós comunicantes. Apesar de apresentar uma solução excelente no quesito segurança, considerando as características limitantes dos sensores, as necessidades de armazenamento tornam essa abordagem proibitiva para redes de sensores com muitos nós.

Considerando esses dois casos extremos, o problema do gerenciamento de chaves pode ser dividido em seis subproblemas (HU e SHARMA, 2005): a pré-distribuição de chaves; o descobrimento dos nós vizinhos que compartilham chaves em comum; o estabelecimento do caminho de chaves entre os nós fins de uma comunicação; o tratamento do caso de nós isolados; o problema da redistribuição de chaves; e por fim a latência dispendida num esquema de distribuição de chaves.

3. REDES WIRELESS

3.1. ROTEAMENTO EM REDES AD HOC

O ambiente de redes sem fio pode ser dividido em dois grupos distintos. Os que possuem alguma infraestrutura de rede e aqueles que não possuem nenhuma infraestrutura de rede. As soluções de redes infraestruturadas geralmente estão baseadas em um ponto de ligação com uma rede cabeada através de um ponto de acesso ou uma estação base. Já nas redes sem essa infraestrutura, os nós podem se comunicar diretamente entre si, sem a necessidade da atuação de um ponto de acesso ou de uma estação base.

3.1.1. Redes ad hoc

Uma rede ad hoc é formada por um conjunto de nós móveis interligados de maneira espontânea num ambiente sem fio. Como os nós tem total liberdade de movimento, a topologia da rede pode mudar constantemente e de forma imprevisível.

Devido a não utilização de infraestrutura de redes cabeadas e pela abrangência de transmissão restrita, a comunicação em redes ad hoc pode ser dividida em outros dois grupos. A comunicação direta e a comunicação utilizando políticas de múltiplos saltos. Na comunicação direta, tanto o nó origem quanto o destino estão dentro do alcance de transmissão um do outro. Este tipo de comunicação está ilustrado na figura 8.

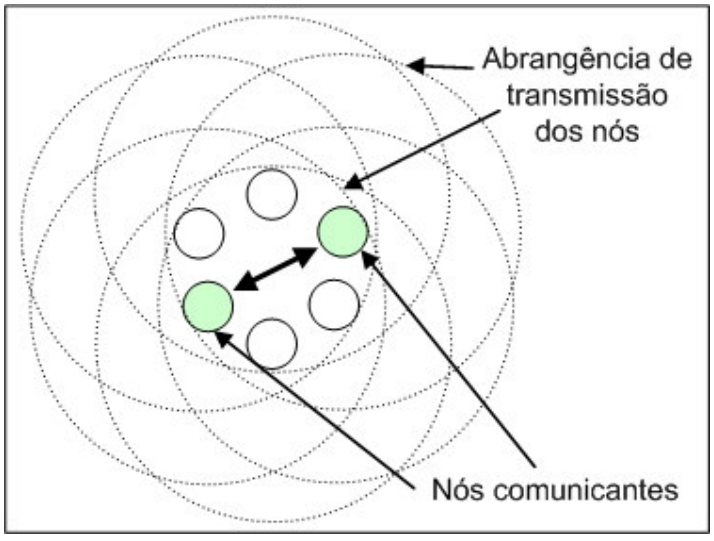

Figura 8: Comunicação direta numa rede ad hoc

Quando um nó de origem precisa se comunicar com outro nó destino que não está dentro da área de abrangência da sua transmissão, a comunicação precisa se dar com a ajuda dos nós intermediários entre eles. Este nós funcionam como roteadores, recebendo e repassando as mensagens provenientes da comunicação entre os nós origem e destino. Este tipo de comunicação é ilustrado na figura 9.

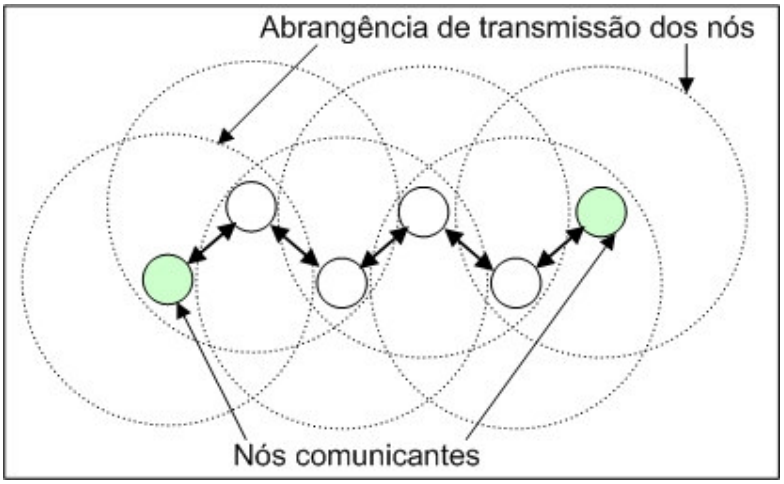

Figura 9: Comunicação de múltiplos saltos numa rede ad hoc

3.1.2. Roteamento pró-ativo e reativo

O foco do trabalho que está sendo proposto se baseia justamente no roteamento de redes ad hoc, quando utilizam políticas de múltiplos saltos para realizá-lo. Neste caso, o roteamento pode ser classificado em reativo e pró-ativo, dependendo da maneira como as rotas são descobertas e armazenadas nos nós.

Os protocolos baseados no roteamento pró-ativo têm como principal característica o momento e a maneira como as informações relativas as rotas são mantidas nos nós da rede. Cada nó possui todas as informações de rota aos possíveis destinos, antes de ser necessário estabelecer uma comunicação.

Desta forma, quando uma mensagem precisar ser enviada para um nó destino, o nó de origem pode transmiti-la diretamente pois já possui a rota completa até o destino armazenada na sua tabela de roteamento.

Os protocolos reativos ou sob demanda se diferem dos pró-ativos na forma como as rotas são estabelecidas. Ao contrário dos protocolos pró-ativos, onde o nó origem já possui o

caminho completo até o nó destino na sua tabela de roteamento, os protocolos sob demanda fazem o descobrimento de rotas somente quando há necessidade de comunicação entre dois nós.

Quando o nó de origem deseja transmitir uma mensagem para o nó destino, primeiro é necessário o estabelecimento da rota de comunicação. Para isso, o nó de origem inicia o envio de mensagens de requisição de rota, que são repassadas até o nó destino ou até um nó que tenha uma rota armazenada com o nó destino. Neste momento, é enviada uma resposta de rota endereçada para o nó de origem. Quando esta mensagem de resposta atinge o seu alvo, finalmente a rota é estabelecida e a mensagem pode ser transmitida.

3.1.3. O protocolo AODV

O protocolo de roteamento AODV (PERKINS e BELDING-ROYER, 2003) é um protocolo reativo, que atua sob demanda e que foi projetado para redes ad hoc móveis. É composto por um conjunto de quatro mensagens, que são enviadas utilizando a porta 654 do protocolo UDP. Cada nó possui uma tabela de roteamento própria, que possui os seguintes campos: endereço IP do destino, número de seqüência do destino, indicador (*flag*) de validade do número de seqüência do destino, indicadores de outros estados e de rotas (válida, inválida, reparável, sendo reparada), interface de rede, contador de saltos (número de saltos necessários para alcançar o destino), próximo salto, lista de predecessores e tempo de vida (tempo de expiração da rota).

Cada nó precisa manter a informação mais atual disponível sobre os números de seqüência para os endereços dos nós destinos em cada entrada da sua tabela de roteamento. Este número de seqüência é tratado como número de seqüência do destino. O correto funcionamento do AODV depende de cada nó da rede para a manutenção dos números de seqüências dos destinos, garantindo assim que as rotas sejam livres de ciclos.

Sempre que um nó recebe uma das mensagens do AODV, proveniente de outro nó, ele verifica a entrada da tabela correspondente ao destino. Se não encontra, cria uma nova entrada. O número de seqüência é determinado pela informação contida no pacote de controle senão o valor falso é atribuído para o campo de número de seqüência válida.

A rota só é atualizada em três casos: se o novo número de seqüência é maior que o número de seqüência do destino armazenado na tabela; se o número de seqüência for igual

mas o contador de saltos recebido adicionado de um for menor que o campo número de saltos armazenado na tabela; e por último, se o número de seqüência é desconhecido.

A descoberta de rotas se dá através da inundação da rede com pedidos de rota por difusão iniciando com o nó que deseja começar a transmissão. Esta mensagem chama-se requisição de rota (RREQ) e seu formato está ilustrado na figura 10.

Este processo de descoberta é feito por todos os nós vizinhos que simplesmente repassam a mensagem recebida para os seus vizinhos, e assim por diante, até que o nó destino receba esta requisição ou que um nó intermediário tenha uma rota já estabelecida na sua tabela de roteamento.

Figura 10: O formato de uma mensagem de requisição de rota RREQ do AODV

Durante o processo de repasse de RREQs, um nó intermediário armazena na sua tabela de roteamento o endereço do vizinho que lhe enviou, por difusão, o primeiro pacote RREQ recebido. Deste modo é estabelecido o caminho reverso. As demais cópias que este nó receber da mesma requisição são descartadas.

Quando o RREQ alcança o nó destino ou um nó intermediário que contém a rota, este nó envia uma mensagem de resposta de rota (RREP) para o vizinho que lhe enviou primeiro o RREQ, o qual repassa o RREP para os nós precursores do mesmo RREQ, até que chegue ao nó de origem da solicitação de rota. O formato da mensagem de resposta de rota é ilustrado na figura 11.

Figura 11: O formato da mensagem de resposta de rota RREP do AODV

A manutenção das rotas é feita através de envios periódicos de mensagens (HELLO) por difusão para os nós vizinhos. Quando um nó se move, ele precisa reiniciar a descoberta de rota para atingir um nó destino. Por outro lado, quando um nó que faz parte de uma rota se move, seu vizinho antecessor no envio da mensagem, irá notar a quebra do link e propagar uma mensagem de erro (RERR) para cada um dos nós anteriores que participam daquela rota. Desta maneira, o nó de origem pode iniciar um novo descobrimento de rota. O formato da mensagem de erro é ilustrado na figura 12.

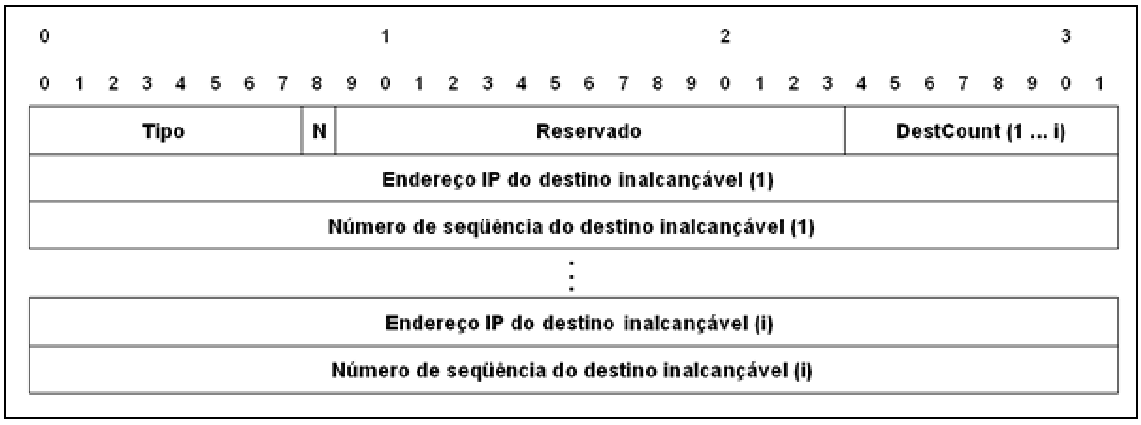

Figura 12: O formato da mensagem de erro de rota RERR do AODV

Uma vez estabelecida a rota entre o nó de origem e o de destino, a mensagem de dados pode ser transmitida. Apesar deste atraso no estabelecimento de rotas para o envio de uma mensagem, o AODV tem a propriedade de se adaptar rapidamente às variações de mobilidade da rede. O principal atrativo é evitar o desperdício de banda e de energia dispendida para a descoberta de rotas que não são necessárias serem detectadas num determinado momento. Como a maioria dos protocolos que atuam sob demanda, no AODV, primeiro deve haver a necessidade de uma determinada rota para esta ser detectada e utilizada.

3.1.4. Ataques possíveis e falhas de segurança do protocolo AODV

Como exposto anteriormente, as características das redes ad hoc fazem dela mais propensa a ataques de segurança. A dependência do correto funcionamento de cada um dos nós que compõem a rede também é uma limitação. Os primeiros protocolos de roteamento desenvolvidos para este tipo de ambiente, que não foram direcionados a área de segurança, apresentam falhas que podem comprometer todo o funcionamento da rede.

Com o protocolo AODV não é diferente. Seu enfoque está na rápida detecção e reorganização dinâmica de rotas, considerando a mobilidade entre os nós comunicantes, provendo escalabilidade e desempenho. Este protocolo foi projetado para reduzir o uso da banda por conta do tráfego de mensagens de controle e o tempo extra gasto para o tráfego de dados.

O foco do protocolo foi o desempenho ao invés de segurança. Os autores do protocolo AODV afirmam que os nós confiam uns nos outros, seja pelo uso de chaves pré-configuradas ou porque é sabido que não há nós intrusos maliciosos (PERKINS e BELDING-ROYER, 2003).

No entanto, se for considerada a possibilidade de nós intrusos maliciosos fazerem parte da rede, eles podem participar do processo de descoberta e manutenção de rotas. As mensagens RREQ, RREP e RERR podem ter alguns campos alterados, trazendo um mau funcionamento ou até mesmo inviabilizando a comunicação na rede. A seguir serão apresentados alguns dos possíveis ataques ao protocolo AODV.

- Alterando o campo: número de seqüência do destino

A alteração do número de seqüência do destino pode acarretar ao redirecionamento do tráfego da rede através de um nó anômalo e até ao impedimento do alcance do destino, que são comportamentos indesejados no descobrimento de rotas. A figura 13 ilustra uma disposição de nós na rede em que pode haver um ataque através da alteração do número de seqüência do destino.

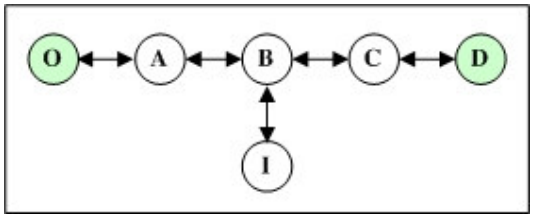

Figura 13: Alterando o número de seqüência do destino

Sejam os nós O, origem e D, destino, que querem se comunicar. Como não é possível se comunicar diretamente, a rota precisa ser descoberta usando os nós intermediários entre eles, no caso, A, B e C. Neste caso, O gera um pedido de rota RREQ e transmite via difusão. A recebe e repassa. Assim é feito sucessivamente pelos nós B e C. Quando o nó de destino D recebe este pedido, ele gera RREP com o seu número de seqüência atual. Considerando um nó intruso I, que envia uma mensagem para B se passando pela mensagem RREP de D. A mensagem enviada pelo intruso possui com um número de seqüência do destino maior do que a original enviada por D. Neste caso, a rota será desviada, sendo que as mensagens enviadas por O para D terão o caminho O-A-B-I, ao invés de terem a rota normal O-A-B-C-D.

- Alterando o campo: número de saltos

Quando o campo número de saltos é alterado, outras anomalias podem ser formadas. Uma delas é referente ao mesmo comportamento abordado anteriormente, quando há alteração no campo número de seqüência do destino. Se o campo número de saltos for repassado por I com um valor menor que o que deveria ser recebido pelo nó B vindo de C, a rota estabelecida é a mesma do exemplo anterior (O-A-B-I). Caso um nó anômalo, que não deseja que uma rota passe por ele, basta aumentar o valor do campo número de saltos para um valor bem alto. Com isso, outra rota alternativa que tenha um valor menor será utilizada.

Mas o resultado que se deseja mostrar neste tipo de ataque está relacionado a formação de ciclos (*loops*), onde uma mensagem não pode consegue atingir o destino, pois fica sendo repassada infinitamente entre os participantes do ciclo. A figura 14 ilustra a formação de um ciclo através da alteração do número de saltos.

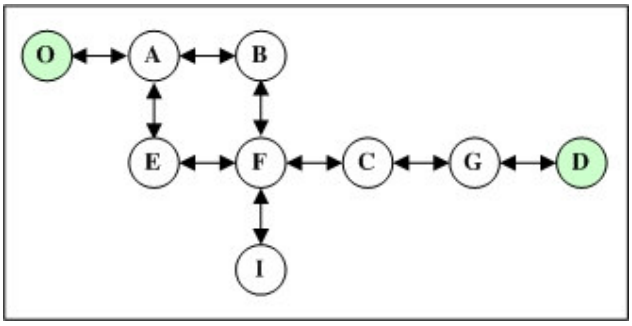

Figura 14: Alterando o número de saltos

Da mesma forma que no ataque anterior, supõe-se que os nós O e D queiram se comunicar. Após o processo de descobrimento de rotas, duas rotas possíveis seriam: O-A-B-F-C-G-D e O-A-E-F-C-G-D. No entanto, se considerarmos a existência de um nó I, intruso,

este pode enviar uma mensagem para F, fazendo-se passar pelo nó E. Além de alterar o campo do endereço da mensagem, o intruso informa a F que a mensagem possui um número de saltos menor do que o informado pelo nó C. Dessa forma, quando uma mensagem for enviada de O para D, passando pela rota O-A-B-F, nesse momento, F repassaria a mensagem para o nó E, que encaminharia para A, formando-se um ciclo e conseqüentemente impedindo que o nó destino D, seja alcançado.

- Forjando mensagens de erro

Como o AODV utiliza mensagens de erro como uma das duas formas de manter as rotas atualizadas (a outra é o envio de mensagens HELLO), a alteração desta mensagem pode fazer com que rotas sejam desfeitas e nós de destino fiquem inalcançáveis.

Usando a figura 14, supondo que um intruso I envie uma mensagem RERR para F se fazendo passar pelo nó C. Esta mensagem enviada contém a informação que a ligação com o nó G foi perdida. Esta mensagem seria repassada através da rota estabelecida, no sentido do nó de origem O, e cada nó estaria atualizando suas tabelas de roteamento e buscando outras rotas alternativas para o nó destino D.

- Outros ataques

Outros ataques possíveis são: a modificação do campo tempo de vida, fazendo com que o nó que receba a mensagem RREP não considere a rota como válida; alteração dos endereços tanto de destino quanto de origem das mensagens RREQ e RREP, onde um nó intruso se faça passar por outro nó da rede; e a própria alteração do valor de qualquer campo ou mesmo bit das mensagens do protocolo, fazendo com que o mesmo tenha um comportamento não desejado ou não previsto.

3.1.5. As extensões do SAODV

Zapata e Asokan (2002) propõem extensões de segurança para proteger as mensagens de roteamento originais do AODV (SAODV). O SAODV usa assinaturas digitais para autenticar os campos não mutáveis e funções *hash* para autenticar as mensagens RREQ e RREP.

Pelo fato dos dois únicos campos mutáveis destas duas mensagens serem o número de saltos e o campo *HASH*, os demais campos podem ser assinados e suas integridade e autenticidade podem ser verificadas fim a fim. Como os nós intermediários alteram o valor do

número de saltos, este não pode ser fixo e incluído na assinatura. Também não pode ter uma assinatura somente para o campo número de saltos, necessitando ser verificada a cada salto, pois tomaria um tempo muito precioso no descobrimento de uma rota.

Para contornar este problema, é utilizada uma função *hash*. Durante o processo de descobrimento de rota, o nó de origem seleciona um número randômico que será utilizado como semente. Ele inicializa o campo número máximo de saltos (*max hop count*) com o valor do campo Tempo de Vida (*time to live*) do cabeçalho IP. Inicializa o valor do campo *Hash* com o valor da semente gerada. Calcula o valor *hash* superior (*top hash*), aplicando a função *hash* x vezes na semente, onde x é o número máximo de saltos.

Esta abordagem é feita para que nenhum intruso decremente o valor do número de saltos da mensagem, fazendo com que ela tenha um comportamento diferente do esperado. Quando os nós intermediários recebem a mensagem, eles aplicam a função *hash* y vezes, onde y é determinado pela subtração dos campos número máximo de saltos e o contador de saltos. O resultado desta função é comparado com o conteúdo do campo *hash* superior e assim comprovada a integridade da mensagem.

Antes de difundir um RREQ ou repassar um RREP, o nó intermediário incrementa o valor do campo contador de saltos em uma unidade e computa o novo valor do *Hash*, aplicando a função *hash* sobre o campo *Hash*. (i.e., *hash*(*Hash*)).

3.2. TRABALHOS CORRELATOS

Os trabalhos relacionados com o escopo deste trabalho podem ser divididos em dois grupos. O primeiro grupo trata do gerenciamento de chaves, enquanto o segundo trata do roteamento seguro em redes ad hoc. Vale ressaltar que algumas das soluções que serão apresentadas foram propostas para redes ad hoc, podendo ser aplicadas a redes de sensores apenas parcialmente, pois nem sempre as limitações que os sensores impõem permitem uma abordagem integral destas soluções.

Dentro do primeiro grupo, que trata do esquema de gerenciamento de chaves, pode-se citar Hubaux et al. (2001), que propõem um repositório de certificados, onde cada nó possui um conjunto de certificados armazenados. Tanto o seu certificado quanto os certificados dos demais nós da rede são processados localmente. Dois problemas podem ser detectados. Primeiro, se o número de nós for grande, o tamanho do repositório necessário ultrapassará a capacidade de armazenamento do sensor. Segundo, se poucos certificados forem armazenados

no repositório, temos um problema de desempenho que pode afetar o funcionamento da rede como um todo.

Muitos trabalhos abordam a criptografia limiar a qual permite associações de nós para que juntos sejam a fonte de confiança de certificados de chaves públicas. Zhou e Hass (1999) propõem o uso de um esquema limiar para distribuir os certificados de chaves públicas através de nós especializados. Cada nó é capaz de gerar uma parte da assinatura compartilhada. Mas somente após a combinação de n partes é que o certificado válido pode ser obtido Qualquer combinação de n-1 partes não conseguem reconstruir o certificado. O principal problema é o tempo necessário para fazer a combinação das partes para poder montar um certificado válido. Infelizmente, devido às limitações dos sensores, todos estes trabalhos envolvendo exclusivamente criptografia assimétrica não podem ser aplicados diretamente neste tipo de ambiente.

Outros trabalhos foram feitos na tentativa de apresentar uma solução viável para uma rede de sensores, considerando as limitações inerentes de um sensor. De acordo com Anjun e Sarkar (2006), uma solução prática seria distribuir as chaves antes dos sensores serem inseridos na rede. Basagni et al. (2001) apresentam uma solução baseada exclusivamente em criptografia simétrica, que provê autenticação de grupo, integridade de mensagens e confidencialidade. Para garantir o sigilo, é proposto ainda um esquema de redistribuição das chaves de tempos em tempos. A desvantagem desta abordagem é que uma vez que o nó é comprometido, o sigilo é revelado, ficando toda a rede comprometida. Os autores propõem uma solução bastante dispendiosa para evitar este tipo de problema, que é a utilização de hardware *tamper resistant*. Segundo Fokine (2002) e Hu e Sharma (2005) esta abordagem é ainda a mais aplicável a uma rede de sensores, porém, com o problema da captura de nós.

Na tentativa de suprir esta deficiência e ainda prover uma comunicação que não esteja focada na comunicação de grupo ou de broadcast, Eschenauer e Gligor (2002) propõem um esquema de distribuição baseada num repositório de chaves. Nesta abordagem, cada sensor recebe um subconjunto de chaves simétricas. De acordo com a distribuição dos sensores, espera-se que pelo menos dois sensores vizinhos compartilhem pelo menos uma chave em comum. Um dos principais problemas desta abordagem é que dois nós vizinhos podem não ter uma chave compartilhada e terem que se comunicar por uma rota que envolva diversos nós adjacentes. Outro problema é o tempo para descobrir qual chave dois vizinhos compartilham dentre muitas armazenadas em seus repositórios.

Dentro desta mesma idéia, diversos autores (OLIVEIRA et al., 2006; CHAN, PERRIG e SONG, 2003; ZHU, SETIA e JAJODIA, 2003; LIU e NING, 2003) desenvolveram seus trabalhos baseados na pré-distribuição de chaves para serem compartilhadas par a par na rede de sensores. Considerando uma rede de sensores estática, talvez esta seja a abordagem mais promissora e uma das mais pesquisadas e aperfeiçoadas. Justamente este é um dos principais problemas deste tipo de abordagem. A topologia da rede precisa ser conhecida antes da sua real implementação. Infelizmente, este modelo de distribuição de chaves não se aplica ao ambiente que se está propondo neste trabalho. Quando se considera a possibilidade de mobilidade entre os sensores, uma abordagem diferente precisa ser adotada.

O segundo grupo de trabalhos relacionado com este trabalho trata do roteamento seguro em redes ad hoc. Perrig et al. (2001) apresentam SPINS, uma arquitetura onde uma estação base acessa os demais nós através de um roteamento baseado na origem. Dois protocolos são apresentados. O SNEP para troca de mensagens entre dois nós e o μTESLA para broadcast, sendo baseados em criptografia simétrica. Nesta arquitetura, a estação base atua como a única entidade de confiança, sendo que cada nó confia somente em si e na estação base, sendo este o fator limitante. Por ser a única entidade de confiança entre os nós, a detecção de rotas depende exclusivamente da estação base. Também a comunicação entre dois nós quaisquer depende da autenticação feita pela estação base.

Zapata e Asokan (2002) propõem uma alteração nas mensagens RREQ e RREP do protocolo AODV, através do uso de assinaturas digitais para garantir autenticidade e integridade dos campos não mutáveis. Para o campo número de saltos de ambas as mensagens, é usado uma função *hash* que permite a cada nó que recebe uma mensagem verificar que este campo não foi reduzido por um intruso. Os autores ainda supõem que há um esquema distribuído de autoridades certificadoras(ACs) onde cada sensor teria acesso, a qualquer momento, para poder autenticar uma chave pública de outro sensor na rede com o qual queira se comunicar. Vale ressaltar que os próprios autores destacam que esta abordagem não se aplica diretamente a dispositivos com limitações tais como os sensores.

Depois de analisar cada tipo de abordagem, acredita-se que o uso de um protocolo híbrido seja bastante aceitável, uma vez que utiliza a agilidade na transmissão pela utilização de chaves simétricas, bem como utiliza as vantagens de flexibilidade e escalabilidade oferecidas pelo uso de chaves públicas. A utilização de chaves públicas se destaca principalmente para ser aplicada no estabelecimento de chaves de grupo, na proteção do segredo de grupo contra captura de nós e para possibilitar a interconexão transparente de *clusters*.

Kamal (2004) propõe o uso de chaves simétricas e *clusters* usando ICP, porém no seu trabalho não é apresentada uma solução consistente nem simulações que comprovem o que foi proposto. O que será apresentado neste trabalho possui algumas características parecidas com as que foram abordadas por Kamal no seu trabalho, tais como utilizar a combinação de chaves públicas com chaves simétricas numa rede de sensores dividida em *clusters*.

4. PROPOSTA DE PROTOCOLO HÍBRIDO

Este trabalho tem o objetivo de apresentar uma solução que garanta a comunicação segura numa rede de sensores sem fio dividida em *clusters*. Para que esta comunicação possa ser feita de maneira transparente entre os sensores de cada cluster, é proposto um esquema de distribuição e gerenciamento de chaves de grupo, que abrange o protocolo para estabelecimento das chaves e uma solução para protege-las, bem como para controlar a redistribuição das chaves de grupo. É prevista ainda a possibilidade de realocação de sensores em outros *clusters*. Também é considerada a hipótese de inserção de um sensor em *clusters* diferentes do que estava destinado.

Após apresentar como as tecnologias de chaves públicas e chaves privadas são utilizadas na solução, o foco do trabalho se dirige para o protocolo de roteamento propriamente dito, onde se deseja que possa funcionar adequadamente no ambiente de *clusters* que está sendo proposto. O grande desafio é fazer o protocolo de roteamento funcionar com o intermédio do *cluster head*, onde as mensagens que são enviadas a outros *clusters* passem necessariamente por ele.

A solução de roteamento é baseada no protocolo AODV, implementando algumas extensões de segurança propostas no SAODV. Tem por finalidade o estabelecimento de rotas e a troca de mensagens de maneira transparente entre os sensores de dois *clusters* quaisquer, sendo necessárias algumas alterações nos protocolos originais que estão sendo utilizados como base (AODV e SAODV).

A seguir são apresentados o modelo e o dispositivo usado como base do desenvolvimento deste trabalho, seguido do esquema de distribuição de chaves propriamente dito e finalizando com a interconexão de *clusters* com transparência.

4.1. DISPOSIÇÃO DOS SENSORES E DOS CLUSTERS

O modelo que é utilizado como base em todo o desenvolvimento deste trabalho é apresentado na figura 15. Pode-se observar o *cluster head* situado no centro do cluster com os demais sensores espalhados aleatoriamente dentro de uma área imaginária. É assumida uma abrangência de transmissão idêntica para todos os sensores dos *clusters*. É importante ressaltar que nesta distribuição aleatória, os sensores podem ser alocados em *clusters* onde o *cluster head* não é o seu *cluster head* supostamente de origem. Cada sensor pode se deslocar

dentro de um cluster e até mesmo entre *clusters*, ultrapassando o limite imaginário da fronteira de um cluster.

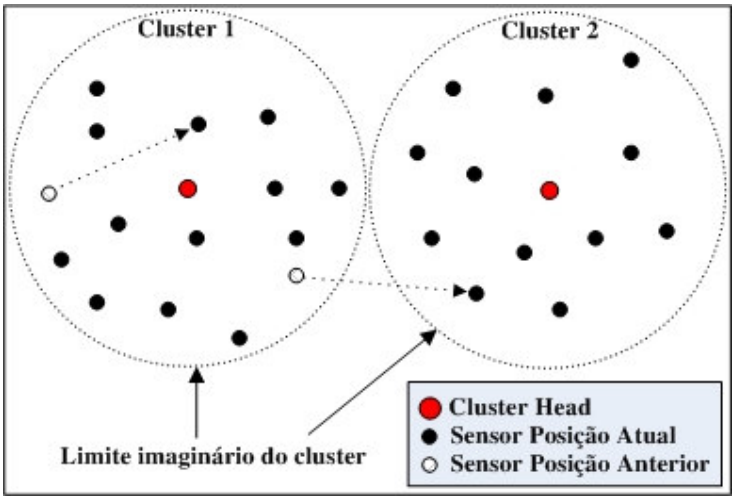

Figura 15: Disposição dos sensores em um cluster

Através da figura 16, observa-se outra particularidade do modelo, no caso com a existência de mais de um cluster, onde não há intersecção entre os *clusters*, ou seja, há uma delimitação física de cada cluster, que pode ser percebida visualmente. A abrangência de transmissão do *cluster head* é maior do que a dos sensores, sendo que cada *cluster head* consegue se comunicar com os *cluster heads* adjacentes a ele. Mas não é grande o suficiente para que os *cluster heads* situados em outros *clusters* não adjacentes possam se comunicar diretamente.

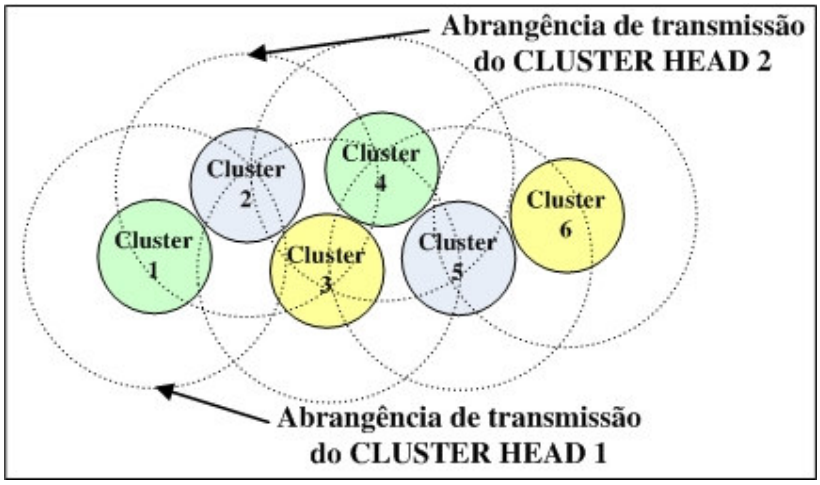

Figura 16: Disposição dos *clusters* de uma rede de sensores

4.2. O ESQUEMA DE DISTRIBUIÇÃO DE CHAVES

A solução apresentada neste trabalho une as vantagens de utilização de chaves simétricas, mais eficientes, com as vantagens oferecidas pela tecnologia de chaves públicas. As chaves simétricas são utilizadas para descobrimento de rotas e para a troca de mensagens entre os sensores de um determinado cluster e chaves públicas para o estabelecimento destas chaves simétricas, tanto na inserção de um novo sensor no cluster quanto na realocação de um sensor em outro cluster.

O esquema de distribuição de chaves obedece algumas premissas básicas. Cada sensor ao ser inserido na rede, possui um conjunto de dados instalados previamente por uma entidade de confiança tanto do sensor quanto do *cluster head*. Estes valores são um identificador próprio (IDsns) e o par de chaves, uma chave privada (KRsns) e outra pública (KUsns). Ainda são pré-instalados no sensor o identificador (IDch) e a chave pública (KUch) do *cluster head* ao qual ele vai pertencer. Esta abordagem se justifica por dois principais motivos. Primeiro, para o sensor é muito dispendioso gerar o par de chaves. De acordo com Malan, Welsh e Smith (2004), o custo para gerar o par de chaves de 163 bits utilizando ECC é de aproximadamente 34 segundos, utilizando a plataforma Mica2 Mote. O segundo motivo é que, caso a chave pública do *cluster head* (KUch) não fosse instalada previamente no sensor, o custo para o estabelecimento de uma relação de confiança entre o sensor e o *cluster head* seria maior ainda.

A distribuição de ACs para que todos os sensores pudessem ter validada uma chave pública recebida seria quase que inviável pelo alcance de transmissão do sensor. Mesmo que esse problema fosse superado, ainda deve ser levado em consideração o modo como a chave de grupo seria estabelecida, pois um dos métodos mais difundidos, o de Diffie-Hellman, é demasiadamente dispendioso para o sensor. Para se ter uma idéia, Malan, Welsh e Smith (2004) estimam que o tempo de troca de chaves usando Diffie-Hellman é de aproximadamente 68 segundos. Recentemente, Blaß e Zitterbart (2005) apresentaram resultados bastante otimistas sobre a implementação eficiente dos pontos de multiplicação dos algoritmos de curvas elípticas. Todo o processo de geração de chaves e de troca de chaves por Diffie-Hellman, passou de aproximadamente 102 segundos (Malan, Welsh e Smith, 2004) para perto de 24 segundos. Mesmo assim, na abordagem deste trabalho, evita-se que o sensor já tenha este gasto prévio, tanto de tempo de processamento quanto de energia de suas baterias.

Como a utilização de uma entidade confiável centralizada para desempenhar o gerenciamento de chaves públicas não é possível num ambiente de redes ad hoc, torna-se necessário encontrar uma solução distribuída (BECHLER et al., 2004). Para viabilizar a utilização de chaves públicas em uma rede de sensores, é preferível que somente um nó em cada cluster utilize uma AC para autenticar as chaves públicas recebidas. E este nó deve ser o *cluster head*, devido as suas características menos restritas. Desta maneira, torna-se muito menor o número de ACs distribuídas necessárias para garantir a disponibilidade desejada.

Para não desviar o foco deste trabalho, será assumido que existe uma distribuição de ACs de modo que a qualquer momento, um *cluster head* pode verificar a autenticidade da chave pública de outro nó com quem ele deseja comunicar-se. Uma outra solução para validação de chaves públicas poderia ser a proposta por Du, Wang e Ning (2005), em que é armazenado um conjunto de *hashes* das chaves públicas dos nós confiáveis.

4.2.1. O protocolo de distribuição de chaves

Para viabilizar o uso de um protocolo criptográfico, os sensores estabelecerão com o *cluster head* uma chave de sessão (simétrica) e a partir desta chave obtida, todas as operações de descoberta de rota e de troca de mensagens serão feitas utilizando esta chave de grupo.

A seguir, na figura 17, será apresentada a seqüência de mensagens que compõem o protocolo de distribuição de chaves.

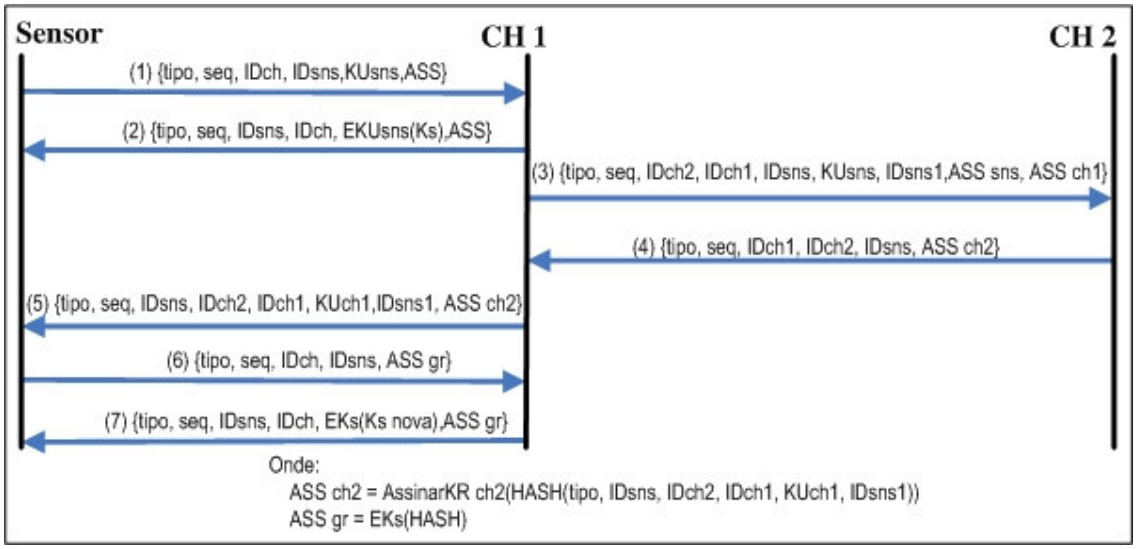

Figura 17: O protocolo de distribuição de chaves

Nesta abordagem, o *cluster head* torna-se a entidade responsável pelo gerenciamento e distribuição destas chaves. Neste trabalho, não se considera que o problema de captura de nós se aplique ao *cluster head*. Para os demais sensores, será proposto um esquema de garantir que a captura dos sensores não seja um problema crucial.

A mensagem (1) representa a solicitação de ingresso no cluster feita pelo sensor, visando a obtenção da chave de grupo. A mensagem (2) representa a resposta de ingresso no cluster feita pelo *cluster head*, com o repasse da chave de grupo. A mensagem (3) representa o encaminhamento para o *cluster head* destino de uma solicitação de ingresso no cluster. Em (4) está representada a resposta de reconfiguração do sensor para reconhecer o novo *cluster head*. A mensagem do tipo (5) fornece os dados necessários para a reconfiguração do sensor para pertencer a um novo *cluster head*. A mensagem (6) representa a solicitação de uma nova chave de grupo e a mensagem do tipo (7) é a resposta à mensagem (6), onde uma nova chave de grupo é fornecida.

Um novo nó ao ser inserido, solicita ao *cluster head* a inserção efetiva no cluster, ou seja, receber a chave de grupo para poder se comunicar secretamente com seus vizinhos. Para isso, ele monta uma requisição que tem o formato (1), onde a Assinatura é uma função de criptografia utilizando a chave privada do sensor, sobre o resultado da aplicação de uma função *hash* sobre os quatro primeiros campos da mensagem. Cada nó ao receber este tipo de mensagem, simplesmente repassa, sem descartá-la, em direção ao *cluster head*, pois somente o *cluster head* pode tratá-la. O *cluster head* ao receber uma requisição de ingresso no cluster, primeiro verifica se a requisição está destinada para ele ou para outro *cluster head*. Adiante será tratado o caso da mensagem ser destinada para o próprio *cluster head*. O caso de endereçamento da requisição para outro *cluster head* será tratado adiante.

Após o *cluster head* receber uma solicitação de inclusão e verificar que a mensagem é destinada a ele, verifica a autenticidade da mensagem usando a chave pública do sensor. A integridade é verificada aplicando a mesma função *hash* sobre os quatro primeiros campos da mensagem e comparando com o *hash* que foi obtido na verificação da assinatura.

Para responder a esta requisição e enviar para o sensor a chave de grupo (Ks) para comunicação, o *cluster head* seleciona a chave de grupo atual – caso ainda não tenha, gera uma nova – e a envia para o sensor com o seguinte formato (2). De posse de Ks, o *cluster head* cifra Ks usando a chave pública do sensor (EKUsns), de modo que somente o sensor ao qual a mensagem está destinada poderá decifrar utilizando a sua chave privada. Após cifrar a chave de grupo, basta juntar os quatro campos (tipo,IDsns,IDch,EKU(Ks)), aplicar uma

função *hash* sobre os valores e assinar o *hash*, para garantia de autenticidade e integridade. Antes de enviar a resposta para o sensor, o *cluster head* registra o ID do sensor e a chave que foi distribuída numa tabela que relaciona IDs a chaves de grupo. O funcionamento desta tabela será explicado melhor na seção 3.2.2.

Devido à abrangência de transmissão do *cluster head*, as mensagens originadas por ele não precisam ser repassadas através dos nós vizinhos, podendo ser descartadas caso o sensor verifique que o destinatário da mensagem não seja ele.

Ao receber esta mensagem, o sensor de destino verifica a assinatura utilizando a chave pública do *cluster head* que está armazenada em sua memória e depois compara o *hash* dos quatro primeiros campos, com o que foi obtido na verificação da assinatura. Uma vez comprovada a autenticidade e a integridade dos dados, o sensor decifra a chave simétrica utilizando a sua chave privada e assim obtém a chave de grupo, para poder começar a participar efetivamente do cluster.

Este momento de inicialização é o que dispende maior tempo de processamento, comunicação e energia para o sensor. Uma vez obtida a chave de grupo, toda a comunicação dentro do cluster é feita utilizando criptografia simétrica.

Quando o *cluster head* CH1 de um determinado cluster C1 recebe uma solicitação do tipo (1) e verifica que o endereço de destino não é igual ao seu, ele encaminha uma mensagem (3) para o *cluster head* destino CH2 do cluster C2, comunicando que há um sensor, que deveria pertencer ao cluster C2, solicitando ingresso no cluster C1. Em vez de CH1 fazer a verificação da assinatura de (1), ele incorpora a assinatura em (3), juntamente com o IDsns do sensor solicitante e o novo ID que o sensor receberá (IDsns1), assina a mensagem, e envia para CH2. Ao receber a mensagem do tipo (3), CH2 primeiro verifica a autenticidade e a integridade da mensagem. Depois, ele monta a mensagem utilizando os campos ID ch2 e IDsns, juntamente com a assinatura enviada pelo sensor e o tipo correspondente a uma mensagem de ingresso no cluster (1). Feito isso, a assinatura do sensor pode ser verificada, assim como a integridade da mensagem. Caso o *cluster head* destino CH2 não estija no seu alcance de transmissão, os *cluster heads* adjacentes que estiverem entre esses dois *clusters* farão o repasse das mensagens tanto de requisição quanto de resposta.

Antes dessa requisição ser atendida, o sensor precisa ser reconfigurado para identificar CH1 como seu novo *cluster head*. Para isso, quem deve mandar uma mensagem de reconfiguração é CH2, o qual neste momento é ainda o único com quem o sensor mantém

uma relação de confiança, ou seja, consegue verificar assinatura, por ter sua chave pública e IDch pré-instalados.

Para gerar a mensagem de reconfiguração, o *cluster head* monta (4), com uma particularidade. A assinatura é feita como se os valores de IDch1 e KUch1 estivessem na mensagem. Depois, como CH1 possui sua identificação e sua chave pública, não é necessário envia-los para CH1. Esta assinatura é importante, pois será a assinatura que o sensor reconhecerá quando a mensagem de reconfiguração (5) chegar até ele. Quando CH1 for repassar a mensagem para o sensor, ele simplesmente remonta a mensagem (4), colocando o destinatário como CH2, bem como o IDch1 e KUch1, que serão os dados do novo *cluster head* que o sensor participará. Como a assinatura já foi feita em (4), CH1 simplesmente coloca a mesma assinatura e repassa (5) para o sensor. Desta forma, é possível verificar a autenticidade e a integridade. Depois de tratar a mensagem, o sensor terá posse de IDch1 e KUch1, podendo assim fazer uma solicitação inicial de chave de grupo(1), como já foi explicado anteriormente.

No entanto, para total realocação e reconfiguração do sensor duas medidas precisam ser tomadas. Uma, quando o CH2 repassa a mensagem de reconfiguração para CH1, ele deve também informar a CA o novo ID que o sensor terá para fazer a relação de KUsns e IDsns. A outra medida a ser tomada é quando o sensor receber a mensagem de reconfiguração, ele precisa mudar o seu IDsns, pois o seu endereço no novo cluster será diferente do antigo.

Geralmente, um sensor solicita somente uma vez a inclusão no cluster. Outra solicitação de inclusão pode ocorrer somente em três casos. Primeiro se o sensor se move para outro cluster, em que os sensores a sua volta trocam mensagens criptografadas utilizando outra chave de grupo. Esta situação será abordada a seguir. O segundo caso de solicitação de inclusão ocorre se o sensor estiver sob ataque, não conseguindo decifrar as mensagens que está recebendo, usando sua chave simétrica. O terceiro caso ocorre, se a bateria do sensor já estiver com nível baixo de energia, onde numa oscilação os dados que estavam em memória podem ser perdidos. Neste caso, porém, o *cluster head* pode ignorar este sensor, pois a chance de falhar é extremamente alta.

Quando um sensor que já possui chave de grupo move-se de um cluster para outro, torna-se necessário a obtenção de uma nova chave de grupo. Deste modo, o sensor vai precisar enviar uma mensagem de inclusão no cluster do tipo (6), que se difere da mensagem (1) somente porque a assinatura é cifrada com a chave de grupo que o sensor possui naquele momento, passando a ser uma assinatura de grupo e não individual. Assume-se, porém, que

pela relação de confiança estabelecida na obtenção da chave de grupo, confia-se que o sensor seja mesmo quem ele diz ser. Os passos seguintes são basicamente os mesmos dos apresentados anteriormente quando um sensor é inserido num outro cluster. A única diferença é que o *cluster head* designado para verificar a assinatura terá que decifrar usando a mesma chave de grupo que está na tabela que relaciona ID de sensor com chaves de grupo distribuídas. E também que a assinatura do *cluster head* será feita utilizando a mesma chave de grupo que o sensor em questão já possui. Esta mensagem tem o formato do tipo (7) do esquema da figura 14. O uso desta tabela de referência é explicado melhor na seção 4.2.2.

Em todos os casos anteriores, se a integridade ou a autenticidade de uma mensagem não possa ser comprovada, o nó que recebeu descarta a mensagem. Outro dispositivo que auxilia na segurança das mensagens é o número de seqüência da mensagem, que identifica a mensagem que está sendo enviada. Tanto o nó emissor quanto o receptor guarda as últimas seqüências enviadas, guardando também o identificador de quem enviou. Esta informação pode ser guardada numa tabela de referência, semelhante a tabela das chaves distribuídas. Este procedimento se faz necessário para evitar ataques do tipo repetição (*replay*), onde o intruso captura uma mensagem e depois de algum tempo, a utiliza para tentar obter uma nova chave ou causar um comportamento não previsto no esquema de distribuição de chaves.

4.2.2. Protegendo e controlando a redistribuição das chaves de grupo

Como a chave simétrica é na verdade uma chave de sessão, estabelecida somente uma vez e sendo mudada de acordo com a política de troca de chaves adotada, a preocupação deve estar concentrada na proteção da memória RAM, barramento e CPU do sensor.

Esta solução ainda não é a ideal, mas se comparada com as soluções apresentadas anteriormente, que se baseiam na proteção completa do sensor via hardware, pode-se concluir que há um avanço neste tipo de abordagem. Se houver um hardware que garanta a proteção do conteúdo da memória RAM, no sentido de que se for tentado fazer uma extração ou leitura do seu conteúdo, seus dados sejam apagados, o esquema de gerenciamento de chaves garante a segurança da rede. Mesmo que o intruso tenha em mãos os dados gravados em disco, podendo acessa-los, que são o ID do *cluster head* e sua chave pública, o ID do sensor e até mesmo a sua chave privada.

Para evitar que um estranho possa obter novamente a chave de grupo, utilizando as primitivas de inclusão no cluster, por possuir a chave privada e o ID do sensor, o protocolo

proposto possui um mecanismo de detecção deste tipo de problema. Toda vez que o *cluster head* receber uma requisição de chave de grupo, ele deve consultar uma tabela que guarda os IDs dos sensores que já receberam a chave de grupo. Nesta tabela também consta a chave de grupo que este sensor está utilizando. Caso já haja uma entrada, o sensor só poderá solicitar uma chave usando a primitiva (6), da sessão anterior. O uso de uma tabela de referência entre ID e chaves de grupo é útil também para a redistribuição de uma nova chave, que deve ser feita dentro de um tempo estipulado. O *cluster head* tem o controle de cada sensor que já recebeu e dos que faltam receber a nova chave de grupo.

No caso de um sensor ser realocado em outro cluster, pode ser que quando ele enviar uma requisição para ingressar no cluster, ele o faça usando o modelo da mensagem (6). Como o novo *cluster head* não tem como verificar a assinatura, quem terá que fazer a realocação é o *cluster head* de origem. Mesmo que já tenha havido uma redistribuição de chave de grupo no cluster de origem, o *cluster head* pode obter a chave que aquele sensor está utilizando através de uma consulta na tabela de referência.

4.3. INTERCONECTANDO CLUSTERS COM TRANSPARÊNCIA

Duas das principais críticas sofridas pela abordagem que está sendo feita neste trabalho diz respeito ao problema da captura de nós, que já foi explicado anteriormente como evitar e o outro problema do uso de chaves simétricas em *clusters*, segundo Zapata e Asokan (2002), é que a comunicação fica restrita a um grupo fechado de participantes – os que detêm a chave simétrica.

Recentemente, a criptografia de curvas elípticas vem ganhando um crescente destaque na área de dispositivos limitados como sensores, em relação a outras técnicas de criptografia como o RSA. Dois fatores justificam esta importância. Primeiro o tamanho das chaves, de acordo com o apresentado anteriormente, os algoritmos baseados em ECC utilizam chaves menores do que os baseados em RSA, garantido o mesmo grau de confiabilidade (LENSTRA e VERHEUL, 2001; GURA et al., 2004).

Segundo, que como suas chaves são menores, espera-se que o tempo de processamento e a energia despendida para cifrar e decifrar sejam menores que os do RSA. Não só por isso, mas também pela própria natureza da criptografia ECC que opera sobre um conjunto de pontos numa curva elíptica definida num universo finito. Sua principal operação

criptográfica é a multiplicação escalar sobre um ponto ao invés da empregada no RSA que são operações exponenciais modulares de números inteiros bastante grandes.

Diversos artigos foram publicados no sentido de destacar a aplicabilidade da criptografia de curvas elípticas no ambiente de redes de sensores, dando destaque ao tempo de processamento e consumo de energia reduzidos. Porém, o sucesso do uso de chaves assimétricas num ambiente de redes de sensores vai depender do número de vezes que as chaves de sessão são geradas pelos acordos de trocas de chaves entre os nós fins, como por exemplo, o acordo de Diffie-Hellman (MALAN, WELSH e SMITH, 2004).

4.3.1. O protocolo de roteamento

O protocolo de roteamento abordado neste trabalho foi desenvolvido baseado no protocolo AODV e em algumas extensões de segurança do SAODV. No entanto, houve necessidade de ajustes no protocolo AODV para que o mesmo pudesse funcionar no ambiente proposto.

4.3.1.1. Adequando o protocolo AODV e as extensões SAODV para clusters

Apesar do trabalho de Zapata e Asokan ter sido proposto como uma RFC e conseqüentemente evoluído bastante desde 2002, a primeira proposição do SAODV é a que servirá de base para este trabalho. Isto se deve pelo fato de a solução proposta pela RFC ser bastante genérica, podendo ser aplicada a qualquer tipo de rede ad hoc, diferente do que está sendo tratado neste trabalho, que diz respeito somente a uma rede de sensores.

O protocolo AODV para funcionar de acordo com as proposições deste trabalho, incorporou algumas funcionalidades propostas no SAODV. Primeiro, a inclusão de um campo assinatura, que é baseada em criptografia simétrica, para a descoberta de rota dentro do cluster. Este protocolo AODV modificado é denominado de AODV-IC (Intra Cluster).

Ao invés de utilizar a mesma solução proposta no SAODV, optou-se por utilizar as mesmas mensagens do AODV, só que antes de serem enviadas, são assinadas com a chave de grupo, provendo uma assinatura de grupo. Do mesmo modo, assim que são recebidas, antes de serem tratadas, a assinatura é decifrada pelo receptor e assim podendo ser verificada. Esta abordagem foi adotada por ser a mais eficiente para descoberta de rota dentro do cluster. As mensagens utilizadas pelo AODV-IC são:

RREQ_IC = (tipo, número_saltos, identificador_broadcast, endereço_destino, número_seqüência_destino, endereço_origem, número_seqüência_origem, *timestamp*, assinatura)

RREP_IC = (tipo, número_saltos, endereço_destino, número_seqüência_destino, endereço_origem, tempo_vida, *timestamp*, assinatura)

A segunda adequação necessária no protocolo AODV foi a inclusão de campos para garantir segurança das mensagens trocadas entre os *cluster heads*, sendo denominado de AODV-EC (Extra Cluster). O funcionalidade desses campos é a mesma que já foi abordada na sessão 2.1.4.5. As mensagens utilizadas pelo AODV-EC são:

RREQ_EC = (tipo, número_saltos, identificador_broadcast, endereço_destino, número_seqüência_destino, endereço_origem, número_seqüência_origem, *timestamp*, chave_púbica, *top_hash*, assinatura, *hash*)

RREP_EC = (tipo, número_saltos, endereço_destino, número_seqüência_destino, endereço_origem, tempo_vida, *timestamp*, chave_pública, *top_hash*, assinatura, *hash*)

O protocolo proposto ainda teve a inclusão de uma nova mensagem, chamada de RREQ-INI (Iniciador de Requisição) que possui o seguinte formato:

RREQ_INI = (tipo, identificador_broadcast, endereço_destino, endereço_origem, assinatura)

Isto se deve ao fato de que o protocolo AODV não funcionaria adequadamente caso o *cluster head* destino, ao receber um pedido de rota RREQ_EC, solicitasse internamente um RREQ_IC. Todos os nós receberiam a mesma mensagem, devido a sua abrangência de transmissão e isso definitivamente afetaria na construção das tabelas de roteamento dos nós, bem como na criação da mensagem de resposta de rota RREP_IC, criada pelo nó destino, que destinaria esta mensagem diretamente para o *cluster head*, não conhecendo a existência de sensores intermediários, quando fosse o caso.

Para contornar este problema, o *cluster head* sinaliza para o sensor destino solicitando que ele inicie a descoberta de rota até o *cluster head*. Esta sinalização é o envio da mensagem RREQ_INI, que contém o ID do *cluster head* e o ID do sensor destino. Desta forma, o sensor destino inicia uma descoberta de rota até o *cluster head*, fazendo difusão de RREQ_IC. Ao receber o RREQ_IC, o *cluster head* já sabe a qual distância (número de saltos) se encontra o sensor e sabe que existe uma rota até ele.

Ao responder esta solicitação de rota (RREP_IC), apesar da abrangência de transmissão, o *cluster head* direciona a mensagem para o sensor mais próximo, ou seja, aquele que lhe enviou a RREQ_IC. Enquanto a mensagem RREP_IC é encaminhada para o sensor destino e as tabelas de roteamento são ajustadas e definidas, o *cluster head* dá continuidade com o estabelecimento de rota com o *cluster head* de origem, enviando uma reposta de rota (RREP_EC) e fazendo a multiplicação de Diffie-Hellman para determinar a chave de sessão entre os dois *clusters* comunicantes. É importante ressaltar que assim como todas as mensagens do AODV-IC, a mensagem RREQ_INI também possui uma assinatura de grupo, isto é, o *hash* gerado da mensagem é cifrado com a chave de grupo do cluster.

As duas principais diferenças das extensões do AODV-EC em relação ao SAODV proposto por Zapata e Asokan (2002) é que está sendo incluída na mensagem a chave pública do originador da mensagem. No caso da RREQ_EC, a chave pública do *cluster head* de origem. No caso da RREP_EC, a chave pública do *cluster head* destino da comunicação, que é o originador da mensagem.

Resultados preliminares das simulações acusaram um tempo excessivamente alto para a computação das funções *hash* com intuito de garantir a integridade do campo número de saltos. Com base nestes resultados, foi preferido o envio das requisições e respostas de rota (RREQ_EC e RREP_EC) antes de verificar a integridade tanto da assinatura quanto do campo número de saltos.

Caso um campo tenha sido alterado maliciosamente, o nó receptor ao verificar que a integridade da mensagem foi violada, simplesmente descarta a mensagem, sem utilizar os valores da mensagem para construção da tabela de rotas. Devido à utilização desta abordagem, um mecanismo de detecção de intrusão diferente deve ser adotado. Como não é o enfoque deste trabalho, supõe-se que há um mecanismo desenvolvido para tratar este tipo de anomalia.

Outro fator que deve ser considerado é que as mensagens das extensões do AODV com assinaturas duplas também não foram implementadas.

4.3.1.2. O processo de estabelecimento de rotas de comunicação

O processo de estabelecimento de rotas de comunicação entre *clusters* segue uma seqüência definida de mensagens. A figura 18 ilustra estes passos necessários, tomando por

exemplo três *clusters* adjacentes, em que um sensor origem situado no cluster de origem deseja estabelecer uma rota até o sensor destino situado no cluster destino.

Cada cluster possui um *cluster head*, sendo que o cluster intermediário está representado somente pelo seu *cluster head*, pois os sensores, apesar de receberem as mensagens enviadas de um cluster para outro, não as tratam, pois não são destinadas a eles. Pode ser que haja outros *clusters* intermediários, mas as operações relacionadas com o repasse de mensagens são as mesmas, não havendo necessidade de estes serem considerados como ilustração.

Nos *clusters* comunicantes, há a possibilidade de outros sensores intermediários. Da mesma forma como no caso de *clusters* intermediários, os sensores intermediários também desempenham a mesma função do que está sendo ilustrado como sensor intermediário. Isto vale tanto para o cluster de origem quanto para o de destino.

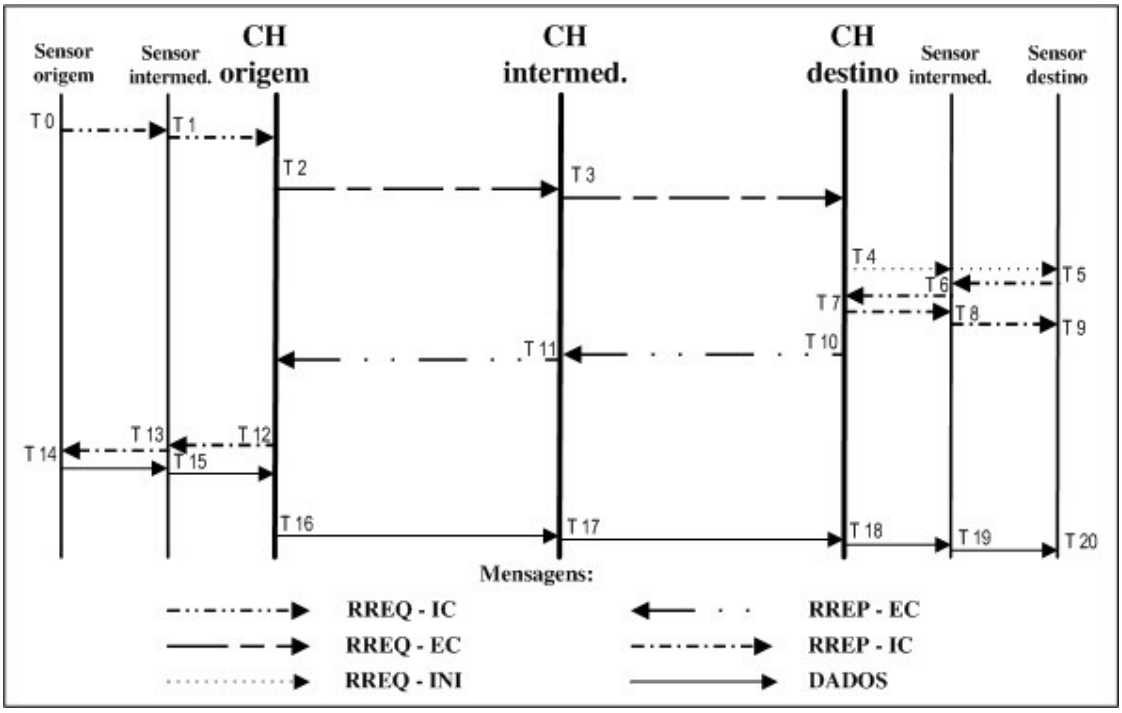

Figura 18: seqüência de mensagens para estabelecimento de rota e envio de dados

Quando o sensor de origem recebe alguma mensagem de dados para transmitir para o sensor destino, ele verifica se existe uma rota definida até aquele sensor. Como não há uma rota estabelecida, ele armazena a mensagem de dados e inicia o processo de descobrimento de rota.

O primeiro passo é enviar uma requisição de rota dentro do cluster. Para isso ele monta a mensagem RREQ_IC. Antes de enviar esta mensagem via difusão (*broadcast*), é gasto um tempo ilustrado na figura como T0, relacionado com o tempo necessário para a aplicação de uma função *hash* sobre os dados da mensagem e o tempo para cifrar este *hash*, gerando uma assinatura de grupo, pois trata-se de uma criptografia simétrica utilizando a chave de grupo.

O sensor intermediário recebe esta mensagem e verifica a assinatura. Como não possui uma rota para o sensor destino, ele incrementa o campo do número de saltos e repassa esta mensagem via difusão. Antes, porém, assina a mensagem. O tempo T1 representa o tempo despendido na verificação da assinatura e na geração da nova assinatura de grupo. Este processo é semelhante caso haja mais sensores intermediários. É importante ressaltar que não estão sendo levados em consideração os tempos necessários para a transmissão da mensagem no espaço (propagação do sinal).

Quando o *cluster head* de origem recebe esta mensagem, ele verifica a assinatura e constata que trata-se de uma requisição de rota para um sensor externo ao cluster. Sendo assim, esta mensagem RREQ_IC precisa ser convertida numa RREQ_EC, onde são ainda acrescentados os campos *top_hash*, o *hash* do campo número de saltos, a chave pública do *cluster head* de origem e a assinatura digital da mensagem, sendo esta baseada em criptografia assimétrica.

O tempo T2 é formado pela soma do tempo de verificação da assinatura de grupo da mensagem RREQ_IC, com o cálculo do *hash* do campo número de saltos, mais o cálculo do valor *hash* da mensagem utilizado na confecção da assinatura digital, juntamente com o cálculo do valor de *top_hash*. Destes valores que compõem T2, o mais dispendioso é relacionado à assinatura da mensagem. Depois, o cálculo do campo *top_hash* pode ser também dispendioso, pois está diretamente ligado ao número máximo de saltos que a mensagem a ser transmitida pode ter.

O *cluster head* intermediário que recebe esta requisição (RREQ_EC), simplesmente atualiza o valor do campo número de saltos e calcula o *hash* deste campo. Portanto, T3 está relacionado somente com este tempo de cálculo da função *hash*. Após repassar a mensagem é que é feita a verificação de sua autenticidade, somente para efeito de construção da tabela de rotas. Se houver outro cluster intermediário, este executará a mesma seqüência de operações.

Quando o *cluster head* destino recebe a requisição RREQ-EC e verifica que é destinada a algum sensor pertencente ao seu cluster, ele primeiro verifica a autenticidade da

mensagem. Após concluir que a mensagem é autêntica, ele monta uma mensagem RREQ_INI e envia para os membros do seu cluster. O tempo T4 está relacionado com estas operações, ou seja, o cálculo do *top_hash* da mensagem RREQ_EC, o cálculo do *hash* da mensagem, a verificação da assinatura, que é a parte mais dispendiosa, e por último a assinatura de grupo da mensagem RREQ_INI. A requisição recebida não é descartada, pelo contrário, é armazenada para poder ser respondida caso uma rota com o sensor possa ser estabelecida.

Todos os sensores que estão na área de abrangência do *cluster head* recebem esta mensagem. Àqueles aos quais a mensagem não é destinada, simplesmente a descartam, e somente o sensor destino responde esta indicação RREQ_INI. Os passos são semelhantes aos feitos pelo sensor de origem, com a única diferença de tentar estabelecer uma rota com o *cluster head* destino. Portanto o tempo T5 é equivalente ao tempo T0 para enviar uma requisição RREQ_IC.

Da mesma forma que no início do descobrimento da rota no cluster de origem, o sensor intermediário que recebe esta RREQ_IC, também efetua as mesmas operações executadas no cluster de origem pelo sensor intermediário, necessárias para repassar a requisição adiante. Neste sentido, os tempos T6 e T1 são equivalentes.

Quando o *cluster head* recebe esta requisição de rota RREQ_IC destinada a ele, é necessário executar duas tarefas distintas. Uma delas é responder a esta requisição, gerando RREP_IC, e onde T7 é o tempo relativo à verificação da assinatura da requisição e a geração da assinatura de grupo da resposta RREP_IC. A outra tarefa é verificar se há alguma requisição RREQ_EC armazenada onde o destinatário seja o mesmo do originador desta RREQ_IC. Esta tarefa deve ser executada também pelo *cluster head* origem, quando recebe uma requisição RREQ_IC. Este passo não foi descrito anteriormente para não atrapalhar o entendimento do protocolo.

Caso o *cluster head* verifique que há uma RREQ_EC armazenada, ele precisa responder esta requisição utilizando o campo número de saltos fornecido pela RREQ_IC e colocando na mensagem de resposta RREP_EC. O tempo T10 é semelhante ao tempo T2, pois as mesmas operações são necessárias para a composição da mensagem RREP_EC. É importante ressaltar ainda que as mensagens RREP_(IC,EC) são sempre endereçadas ao nó que enviou primeiro a requisição correspondente a qual se está respondendo, diferentemente das requisições que são enviadas via difusão.

No repasse desta mensagem, o *cluster head* intermediário executa as mesmas operações necessárias à requisição RREQ_EC. Portanto, o tempo T11 é semelhante ao tempo

T3. Da mesma forma citada na requisição, se houver mais *clusters* intermediários, as operações desempenhadas por eles serão exatamente as mesmas.

Quando o *cluster head*, situado no cluster que originou a requisição RREQ_IC, recebe a resposta de rota RREP_EC, ele precisa transformar esta mensagem de resposta numa RREP_IC. Antes porém, precisa assegurar a autenticidade da mensagem recebida. Este tempo T12 é semelhante ao tempo T4. Somente depois de verificada a autenticidade é que a mensagem RREP_IC é montada e enviada para dentro do cluster.

Os tempos T12 e T4 são semelhantes, porém há uma pequena diferença entre eles. Diz respeito justamente a criação da assinatura de grupo das mensagens RREQ_INI (no caso de T4) e de RREP_IC no caso de T12, sendo esta última maior que a primeira, sendo necessária a divisão da RREP_IC em dois blocos, ao invés da RREQ_INI que pode ser calculada um bloco somente para a execução da função *hash*.

Mesmo após enviar a mensagem de resposta para dentro do cluster, o *cluster head* de origem inicia o cálculo da chave de sessão, definida pelo acordo de Diffie-Hellman, para isso utilizando a chave pública do cluster destino fornecida na mensagem RREP_EC.

A função dos sensores intermediários para repassar uma mensagem RREP_IC já foi descrita anteriormente, no caso do cluster destino. O tempo T13 é semelhante ao tempo T8. Por fim a mensagem de resposta de rota chega ao sensor de origem. Este, quando recebe este tipo de mensagem, verifica a autenticidade de grupo da mensagem.

Após estas operações, o sensor consulta se há algum dado a ser transmitido e, mais importante, se o pacote de dados possui um tempo de vida ainda válido. Se houver, todo o percurso da mensagem será feito usando chaves simétricas. Primeiro, o sensor cifra a mensagem de dados usando a chave de grupo. O *cluster head* ao receber, decifra com a chave de grupo e cifra com a chave de sessão que foi estabelecida com o outro *cluster head*.

Se houver *clusters* intermediários, os *cluster heads* simplesmente repassam as mensagens. O *cluster head* destino ao receber a mensagem, decifra usando a chave de sessão e cifra usando a chave de grupo do cluster destino. A última etapa consiste em enviar a mensagem para o sensor destino e este de posse da chave de grupo do cluster, decifra a mensagem que lhe foi destinada.

Este processo descrito acima será detalhado nos parágrafos seguintes. Quanto o sensor de origem verifica que possui um pacote a ser transmitido, ele utiliza a rota estabelecida para repassar a mensagem. Os dados ainda são cifrados com a chave de grupo. O tempo T14 está

relacionado com a verificação da autenticidade da assinatura contida na RREP_IC e do tempo envolvido para cifrar a mensagem de dados.

O sensor intermediário simplesmente repassa a mensagem adiante, seguindo a rota. Portanto T15 é praticamente nulo, pois nenhuma operação é necessária neste caso. Quando a mensagem chega ao *cluster head* de origem, este precisa decifrar a mensagem de dados usando a chave de grupo e cifrá-la novamente usando desta vez a chave de sessão compartilhada com o *cluster head* destino.

Como o *cluster head* precisa desta chave para poder repassar a mensagem de dados, é necessário que espere a conclusão do cálculo da mesma através da matemática de DH. Isto se torna evidente pelas ordens de grandeza envolvidas nos tempos de processamento das operações criptográficas simétricas e assimétricas. Desta forma, T16 é a soma dos tempos necessários para decifrar e cifrar a mensagem de dados e principalmente pelo tempo necessário para o cálculo da chave de sessão.

O *cluster head* intermediário simplesmente repassa a mensagem adiante, sendo que o tempo T17 é praticamente nulo, da mesma forma que T15. Ao receber a mensagem de dados, o *cluster head* destino precisa fazer a operação inversa da feita pelo *cluster head* origem. Primeiro decifra a mensagem com a chave de sessão e depois cifra com a chave de grupo, enviando posteriormente em direção ao sensor destino. O tempo T18 é bastante inferior ao tempo T18, justamente por ser a soma dos tempos necessários para decifrar e depois cifrar novamente a mensagem de dados.

O sensor intermediário simplesmente repassa, da mesma forma pelo sensor intermediário do cluster de origem, neste sentido o tempo T19 é semelhante aos tempos T15 e T17. O sensor destino, quando recebe a mensagem de dados, simplesmente precisa decifrá-la usando a sua chave de grupo, para depois fazer uso dos dados recebidos.

O mais atrativo nesta abordagem é a forma como os nós comunicantes fazem a comunicação dos dados. Na visão deles, não interessa se o sensor com quem desejam se comunicar esteja no mesmo cluster ou num outro cluster distante, pois para ele tanto a descoberta de rota quanto o envio da mensagem de dados serão feitos da mesma forma, usando a sua chave de grupo.

Da mesma forma o sensor destino responde por pedidos de rota e recebe dados usando sempre a sua chave de grupo. O que interfere é justamente o tempo necessário para estabelecer uma rota entre dois *clusters*, pois descobrimento de rotas no mesmo cluster envolve somente operações criptográficas simétricas. Assim se tem uma interconexão

transparente entre *clusters*, pois para os sensores comunicantes é como se ambos fossem membros de um mesmo grupo que compartilha a mesma chave de grupo.

Ainda pode ser ressaltada a facilidade de estabelecimento de rotas quando qualquer outro sensor de um dos dois *clusters* tentar se comunicar com outro sensor do outro cluster. O processo de descobrimento de rotas será otimizado pois o processo mais dispendioso já foi feito anteriormente, ou seja, já houve o acordo de troca de chaves entre os *cluster heads* e uma chave de sessão já foi estabelecida.

Somente as rotas dentro de cada *cluster* precisam ser estabelecidas. A comunicação entre os clusters pode ser feita usando a mesma chave de sessão já estabelecida. Claro que para isso supõe-se que a chave de sessão entre os *cluster heads* ainda não tenha expirado. Esta parte não chegou a ser desenvolvida neste trabalho.

5. RESULTADOS EXPERIMENTAIS

As simulações têm seu foco principal no desempenho do protocolo proposto para a interconexão de *clusters*, sendo necessária a montagem do ambiente em um software de simulação. Devido a grande aceitabilidade no meio científico, por ter seu código totalmente aberto, que possibilita a modificação e extensão de certas funcionalidades o NS-2 foi o simulador escolhido. Além da definição do ambiente de simulação, o protocolo proposto precisa ser implementado e os cenários de simulação determinados. A seguir serão apresentados os itens relacionados com a execução das simulações, bem como apresentados os resultados obtidos através destas simulações no final da sessão.

5.1. O *NETWORK SIMULATOR* 2 – NS2

O NS-2 é um simulador de eventos discretos orientado a objetos, desenvolvido pela Universidade de Berkley e em constante desenvolvimento e aperfeiçoamento. Neste trabalho foi utilizada a versão 2.29 do simulador.

Este simulador permite simulações de ambientes não só cabeados mas também sem fio, que é o escopo deste trabalho. Ele é escrito em C++ e orientado a componentes, com um interpretador OTcl. Os principais componentes implementados em C++ possuem uma interface que pode ser acessada e modificada via OTcl, unindo o alto desempenho da linguagem C++ à facilidade de alterações na configuração das simulações, permitidas pela OTcl. Os objetos criados na linguagem OTcl mantém uma relação direta com os objetos criados em C++, sendo que as alterações feitas aos objetos OTcl afetam os atributos dos objetos definidos em C++.

O simulador possui um escalonador de eventos, que associa um tempo global de simulação para cada evento e um ponteiro para tratar o objeto relativo àquele evento. Além disso, o simulador traz implementadas muitas soluções de roteamento. Uma delas é o AODV, que foi utilizado como base para as simulações.

A configuração dos cenários das simulações são feitas em OTcl e os resultados podem ser gerados em dois formatos de arquivos distintos. Um com extensão .nam é interpretado por um programa auxiliar chamado Network Animator, que permite a visualização dos eventos simulados. Outra extensão possível é a .tr, que significa trace. Neste tipo de arquivo, são associados os tipos de eventos, em quais nós, o tipo de mensagem, etc., e este foi tipo de arquivo escolhido para interpretação dos resultados das simulações. Os detalhes podem ser

visualizados no ANEXO I, onde constam alguns registros de uma simulação feita para interconectar dois *clusters* adjacentes.

5.2. O MODELO DE SENSOR UTILIZADO

Este trabalho utiliza um modelo de sensor como base de configuração e características para todos os sensores abordados. O Mica2 Mote foi desenvolvido na Universidade de Berkeley e fabricado pela Crossbow Inc. O sistema operacional é o TinyOS, desenvolvido também pela Universidade de Berkeley e é um sistema de código aberto exclusivo para sensores. Dentro das características deste dispositivo pode-se destacar o processador de 8-bits com 7.3828 MHz ATmega 128L, sua memória primária (SRAM) de 4 Kb e 128Kb de espaço para os programas (ROM). Este dispositivo ainda possui uma EPROM de 512 Kb e transmissão a 433-MHz de rádio freqüência, com taxa de transmissão de 38.4 k baud. A figura 19 permite uma visualização de um Mica2 Mote.

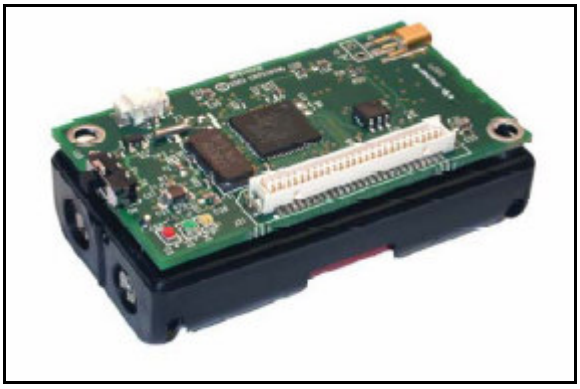

Figura 19: O Mica2 Mote da Crossbow Inc.

O Atmega 128 ainda apresenta algumas particularidades de arquitetura. Ele implementa uma memória de dados homogênea que pode ser acessada por três registros apontadores de 16 bits cada, com as funcionalidades de pré-decremento e pós-incremento. Ainda possui um conjunto de 32 registradores de 8 bits que podem ser utilizados para operações aritméticas. Este processador pode ser operado em freqüências de até 16 MHz, onde um ciclo de instrução se iguala a um ciclo do clock (GURA et. al., 2004).

5.3. A DISPOSIÇÃO DOS SENSORES EM UM CLUSTER

Diferentemente do modelo proposto na sessão de desenvolvimento deste trabalho e para qual o protocolo desenvolvido pode suportar, o modelo de cluster simulado não leva em conta a possibilidade de deslocamento dos sensores, sendo eles estáticos. A figura 20 ilustra bem o modelo utilizado.

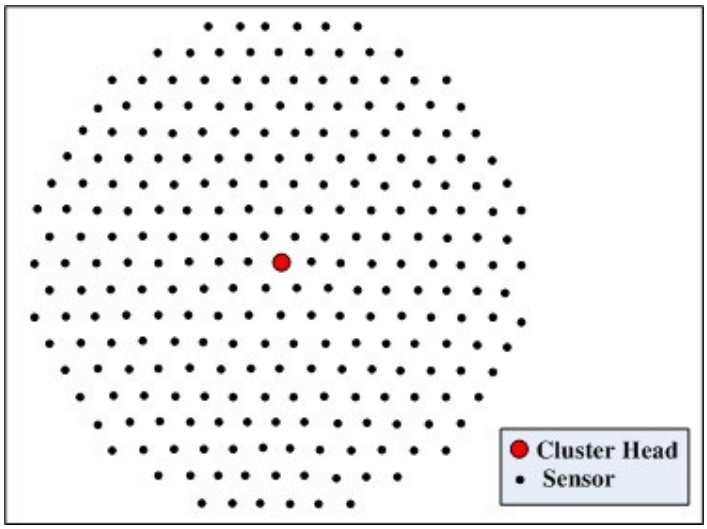

Figura 20: Disposição dos sensores num cluster modelo de simulação

Pode-se notar o *cluster head* situado no centro do cluster com os demais sensores espalhados uniformemente e eqüidistantes uns dos outros. Os sensores estão assim distribuídos para que seja otimizada a ocupação do espaço. Neste sentido, cada sensor consegue se comunicar somente com os sensores adjacentes, ou seja, os que estão imediatamente ao seu lado. Também é assumida uma abrangência de transmissão idêntica para todos os sensores dos *clusters*. Para estas simulações é assumida uma distância de dez metros entre cada sensor.

5.4. A DISPOSIÇÃO DOS CLUSTERS

Os *clusters* estão dispostos no plano, alinhados horizontalmente e os *cluster heads* estão eqüidistantes uns dos outros, a uma distância de 200 metros. Esta disposição é ilustrada na figura 21.

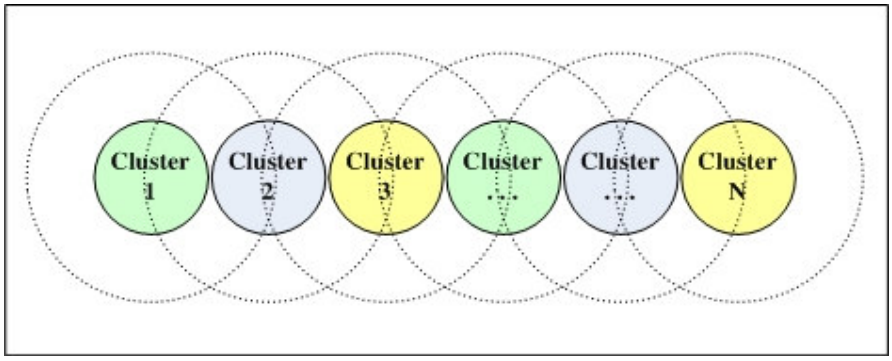

Figura 21: Disposição dos *clusters* no plano

É importante ressaltar que cada cluster que compõe esta figura obedece a mesma disposição da sessão 4.3, ou seja, cada cluster com o seu respectivo *cluster head* situado no centro.

5.5. ALTERAÇÕES NECESSÁRIAS NO NS-2

Para o correto funcionamento do protocolo proposto neste trabalho, é necessária modificação de alguns itens no simulador. O primeiro diz respeito a recompilação do simulador com suporte a Openssl, que precisa estar instalado previamente no computador. A versão utilizada nas simulações foi a 0.9.8d.

Outra alteração necessária diz respeito aos arquivos que se encontram dentro do diretório aodv e trace, que se encontram dentro do diretório ns. Estes arquivos, mais especificamente são: aodv.h, aodv.cc, aodv_packet.h, dentro do diretório aodv e o arquivo cmu-trace.cc dentro do diretório trace. Alguns trechos de código que dizem respeito a estas alterações se apresentam no ANEXO II.

As principais alterações contidas nestes arquivos são a estrutura das mensagens RREQ_IC, RREQ_EC, RREP_IC, RREP_EC, RRINI, que estão contidas no arquivo aodv_packet.h. Já nos arquivos aodv.h e aodv.cc, no primeiro estão a definição dos métodos de recebimento e de envio das mensagens definidas no aodv_packet.h e no segundo a implementação deste métodos, bem como dos métodos de criptografia, que fazem referência as classes implementadas pelo Openssl. Já o arquivo cmu-trace.cc traz as alterações para registrar a troca das mensagens, definidas no aodv-packet.h, no arquivo trace gerado pela simulação.

5.6. AS SIMULAÇÕES REALIZADAS

Como o objetivo das simulações é analisar o desempenho do protocolo para a interconexão de *clusters*, seja para o estabelecimento de rotas quanto para o envio de dados com uma rota já estabelecida, é necessário definir os valores que serão utilizados como métricas de desempenho a serem inseridas no simulador. Também é preciso definir o ambiente a ser simulado, com suas variações. As próximas sessões são dedicadas ao tratamento destas variações.

5.6.1. Os valores utilizados

Devido ao modo como o NS-2 funciona, incluindo atrasos no tempo de simulação para cada operação relevante ao que se quer simular, é necessário um embasamento em outros trabalhos anteriores que apresentam resultados de desempenho das funções criptográficas utilizadas no Mica Motes de 8MHz.

Um dos trabalhos foi o de Ganesan et al. (2003), que determinaram que o custo para se aplicar uma função *hash* SHA-1 sobre 56 bytes é de 3,636 milissegundos e sobre 64 bytes é de 7,777 milissegundos. Também mostram que o custo para aplicar uma cifragem do RC5 sobre 16 bytes é de 0,413 milissegundos, portanto, 1,652 milissegundos se forem considerados 64 bytes. E ainda de 0,826 milissegundos para cifrar os 20 bytes gerados pela aplicação de uma função *hash* SHA-1 e assim garantindo uma assinatura de grupo. O tempo para decifrar é um pouco menor, cerca de 0,409 para 16 bytes de dados, ou 0,818 para os 20 bytes da assinatura de grupo.

Outro trabalho que deve ser considerado para a obtenção dos resultados de simulação foi apresentado por Blaß e Zitterbart (2005). Segundo os autores, o tempo para assinar uma mensagem usando o algoritmo de assinatura ECDSA, baseado em ECC é de 6,88 segundos e de 24,17 segundos para que esta assinatura possa ser verificada. Os autores ainda demonstram que o tempo para a geração de uma chave utilizando o algoritmo de Diffie-Hellman é de 17,28 segundos.

5.6.2. O ambiente simulado

O protocolo proposto é baseado numa rede de sensores dividida em diversos *clusters*, sendo que a cada momento, se deseja estabelecer uma troca de mensagens entre dois sensores quaisquer situados em *clusters* diferentes, sem que haja qualquer rota pré-estabelecida entre eles. Porém, nas simulações, a montagem de cenários para satisfazer todos os sensores posicionados no cluster, torna esta abordagem impraticável.

Para contornar este problema, as simulações visam obter um valor máximo (pior caso) e não o valor médio para o estabelecimento de rota e envio de mensagens de dados. Para isso, os sensores são dispostos de uma maneira que o número de saltos necessários para a definição de uma rota por um sensor seja maior ou igual ao máximo proporcionado pela disposição dos sensores no cluster como abordado na sessão 5.3.

Portanto, não é necessário configurar todo o *cluster* como está sendo apresentado na figura 20. Basta colocá-los de tal forma que as mensagens necessárias para o descobrimento de rotas possam ser trocadas, sem que haja um excesso causado pelos sensores intermediários e que não participam efetivamente do descobrimento de rota.

A figura 22 apresenta a disposição dos sensores, como se fosse feito um corte vertical no *cluster*, com ambos *clusters* opostos um ao outro. Para que o número de saltos seja maior que o número de camadas de sensores apresentadas na figura 20, são colocados 10 sensores alinhados horizontalmente no plano, separados a uma distância de 10 metros.

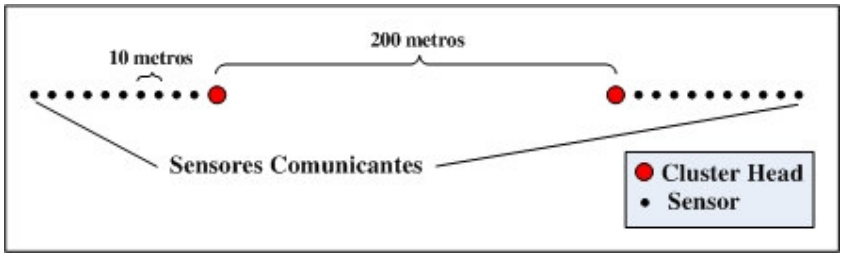

Figura 22: Disposição dos sensores para simulação

Na parte interna dos *clusters* está o seu centro, representado pelos *cluster heads*. Estes estão distantes um do outro em 200 metros, como ilustrado na figura 21, só que omitindo os demais sensores que estariam entre os *cluster heads*.

Como são simulados desde 2 *clusters* (representado na figura 22) até 100 *clusters*, com 98 *clusters* intermediários, estes são representados somente pela figura do seu *cluster head*, que mantém a mesma distância entre *cluster heads* que é de 200 metros.

Como o tipo de sensor que se está usando como base das simulações possui uma característica muito interessante de processar uma instrução a cada clock e duas instruções para operações de multiplicação, pode-se considerar como configuração para o *cluster head* um outro modelo de sensor Mica Motes que opera a 16 MHz [Gura et al., 2004].

Além destas duas velocidades de processamento, é utilizado como suposição a existência de outros dispositivos com a mesma característica do Mica Motes, só que operando nas freqüências de 32MHz e 64MHz respectivamente. Esta suposição ajuda na interpretação dos dados que serão obtidos através da simulação.

Além de simular com quatro freqüências diferentes para os *cluster heads*, também são variados o tamanho dos pacotes de dados transmitidos. Esta variação compreende desde 64 bytes, 128 bytes, 256 bytes, 512 bytes e finalmente 1024 bytes. O tipo de dado utilizado denomina-se CBR (Constant Bit Rate) e é gerado por um agente CBR, que possui como principal característica o envio de mensagens constantes de dados dentro de um determinado intervalo de tempo.

Além de variar o tamanho dos pacotes e a velocidade dos processadores, outro item que se quer enfatizar são os tempos relacionados com a descoberta de uma rota, para o envio de um pacote de dados e com o envio de um segundo pacote de dados, porém com a rota já estabelecida.

Outra característica das simulações deste trabalho é que tanto para conectar dois quanto cem *clusters*, o número máximo de saltos assumido é de 120 saltos. Este dado é relevante, pois determina quantas vezes a função *hash* deve ser aplicada ao campo número de saltos pelo emissor de uma mensagem para poder calcular o valor do campo *top_hash*, tanto das mensagens RREQ_EC quanto RREP_EC.

5.6.2.1. A montagem dos cenários no simulador

Como descrito no início deste capítulo, a montagem dos cenários no NS-2 é feita através de OTcl. Cada cenário deve ser um arquivo com extensão tcl diferente. As principais configurações usadas nas simulações são apresentados na tabela 2.

Tipo de canal	*Channel/WirelessChannel*
Modelo de propagação de rádio	*Propagation/TwoRayGround*
Tipo de antena	*Antenna/OmniAntenna*
Tipo da camada de enlace	LL
Tipo da interface de rede	*Phy/WirelessPhy*
Tipo de camada MAC	Mac/802_11
Protocolo de roteamento	AODV
Tipo de endereçamento	*hierarchical*

Tabela 2: Valores de configuração dos sensores no NS-2

Estes valores são utilizados através do comando:

```
$ns node-config     -adhocRouting AODV \
                    -llType LL \
                    -macType Mac/802_11\
                    -antType Antenna/OmniAntenna\
                    -propType Propagation/TwoRayGround\
                    -phyType Phy/WirelessPhy\
                    -agentTrace ON \
                    -routerTrace OFF \
                    -macTrace OFF \
                    -movementTrace OFF \
                     -channel Channel/WirelessChannel\
                    -addressType hierarchical
```

Todos os nós que forem criados a partir deste comando, irão herdar estas configurações. Antes porém, é necessário determinar os alcances de transmissão de cada sensor. Esta característica é determinada pela classe Phy/WirelessPhy, que possui quatro atributos que precisam ser alterados para que se possa chegar ao cenário desejado.

O atributo Pt_ significa a força da transmissão (*power transmission*). CPThresh_ (*capture threshold*) é o limiar de captura de uma mensagem trafegando pelo espaço. CSThresh_ (*carrier sense threshold*) é o limiar de detecção da portadora que indica o valor mínimo de força de uma transmissão para que esta possa ser detectada. RXThresh_ (*receive threshold)* é o poder (força) mínimo requerido para receber um pacote.

Para configurar os sensores a uma distância de 10 metros, fazendo com que cada um conseguisse transmitir para o seu vizinho imediato, foram utilizados os seguintes valores:

Phy/WirelessPhy set CPThresh_ 10.0
Phy/WirelessPhy set CSThresh_ 8.91e-17
Phy/WirelessPhy set RXThresh_ 1.92252e-13
Phy/WirelessPhy set Pt_ 2.818e-08

Para a configuração dos *cluster heads* deve ser utilizado o valor de poder de transmissão:

Phy/WirelessPhy set Pt_ 9.73276e-04

Estes valores não são determinados ao acaso. Para que os mesmos possam ser definidos, o próprio ambiente do simulador fornece um programa que fica situado no diretório ~ns/indep-utils/propagation/threshold.cc, que deve ser utilizado para a determinação destes valores.

Depois de configurados, os nós representando os sensores podem ser criados. Isto é feito através do comando: set n(0) [$ns node 0.0.1]. Significa que o objeto n(0) vai ser criado com as configurações definidas anteriormente e que este sensor possui um endereço hierárquico 0.0.1. Em seguida, configurada a posição que este sensor vai ocupar no espaço. Os comandos respectivos para este posicionamento são:

$n(0) shape "circle"
$n(0) set X_ 10
$n(0) set Y_ 10
$n(0) set Z_ 0

Com isto, todos os demais podem ser configurados e posicionados de acordo com o modelo apresentado pela figura 21. É importante ressaltar que os sensores tem no seu endereço hierárquico o último valor a partir de 1. Cada cluster é diferenciado pelo segundo campo do endereço hierárquico. Na configuração do endereço dos *cluster heads*, os mesmo possuem o último endereço com valor 0. Exemplo: set n(11) [$ns node 0.1.0]. Este é o *cluster head* do cluster 1.

Depois de configurados e posicionados, é necessário definir o tipo de agente que será utilizado. No caso deste trabalho, o agente CBR.

set sink [new Agent/Null]
$ns attach-agent $n(21) $sink

```
set cbr [attach-CBR-traffic $n(0) $sink 512 100.0]
```

Estes comandos definem um criador e um receptor de mensagens CBR. Neste caso, a terceira instrução está criando mensagens de 512 bytes, que serão enviadas a partir do sensor n(0) e que terão um intervalo de 100 segundos.

A última parte do arquivo tcl é configurar o início e o fim da simulação e executar a simuação propriamente dita. Isto pode ser feito através dos comandos:

```
$ns at 0.0000000001 "$cbr start"
$ns at 105.0 "$cbr stop"
$ns at 200.0 "finish"
puts "Start of simulation.."
$ns run
```

Indica que no tempo 0.0000000001 o agente CBR irá iniciar a transmissão de dados, e no tempo de simulação de 105 segundos irá encerrar o envio de dados CBR. A simulação irá se encerrar no tempo igual a 200 segundos, e por último é dado a instrução de início da simulação.

O arquivo completo utilizado para simular três *clusters* pode ser visualizado no ANEXO III.

Outra consideração importante a ser feita diz respeito a maneira como as simulações foram feitas e os resultados obtidos. Como se tratam de eventos determinísticos, somente uma simulação de cada cenário foi necessária. Isso se justifica pelo que foi explicado anteriormente, que os sensores estão todos eqüidistantes um dos outros e o mesmo acontecendo com os *cluster heads*. Desta forma, como consideramos a transmissão de mensagens sem obstrução (*TwoRayGround*), as mensagens semelhantes possuem sempre a mesma velocidade de propagação.

5.7. OS RESULTADOS OBTIDOS

Os resultados obtidos a partir das simulações foram retirados dos arquivos de trace gerados. Para cada cenário gerado, dois gráficos são gerados. Um diz respeito ao tempo envolvido desde o descobrimento da rota até a entrega do pacote de dados ao sensor destino. O segundo gráfico está relacionado com o tempo gasto para entregar somente um pacote de dados com a rota já estabelecida.

O gráfico 1 apresenta os valores gerados pelo envio de um pacote de 64 bytes de dados CBR sem a rota estabelecida.

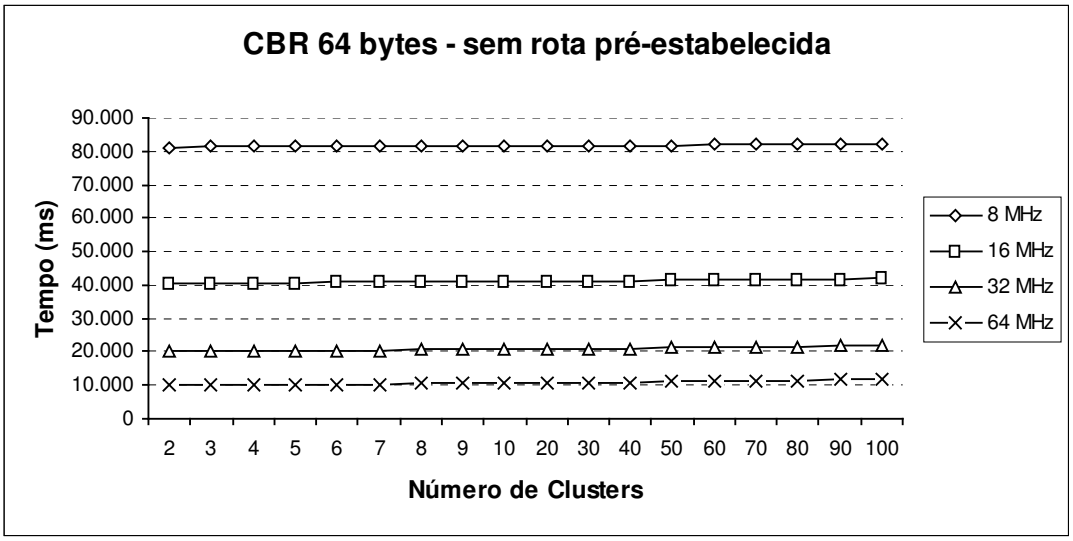

Gráfico 1: Tempo máximo para troca de uma mensagem CBR de 64 bytes sem rota pré-estabelecida

É possível verificar que o desempenho é basicamente o mesmo para cada poder de processamento dos *cluster heads*. Isto se deve ao tempo maior no estabelecimento de rotas ser destinado às operações criptográficas assimétricas. Desta forma, quando os valores do poder de processamento dos *cluster heads* são dobrados, o tempo máximo no estabelecimento de rotas é praticamente reduzido à metade.

O que determina o aumento gradual da interconexão desde dois até cem *clusters* é o tempo de propagação da mensagem no espaço e o tempo de aplicação da função *hash* por cada *cluster head* intermediário. Porém este aumento do tempo global é quase imperceptível, quando comparado a uma ordem de grandeza milhares de vezes maior.

Quando é considerado o tempo máximo para troca de mensagem CBR com a rota já estabelecida, os valores do envio da mensagem caem drasticamente. O gráfico 2 mostra os valores relativos ao envio da segunda mensagem CBR, quando a rota já está estabelecida.

A inclinação das curvas torna-se mais evidente pois o tempo de propagação das mensagens são mais relevantes quando comparados ao tempo total dispendido para o envio da mensagem de dados.

Outro comportamento que é possível ser verificado é a questão do poder de processamento dos *cluster heads*. Ao contrário do gráfico anterior, onde o poder de processamento influencia diretamente no tempo total despendido para o envio da mensagem de dados, no gráfico 2 este poder não se sobressai tão drasticamente.

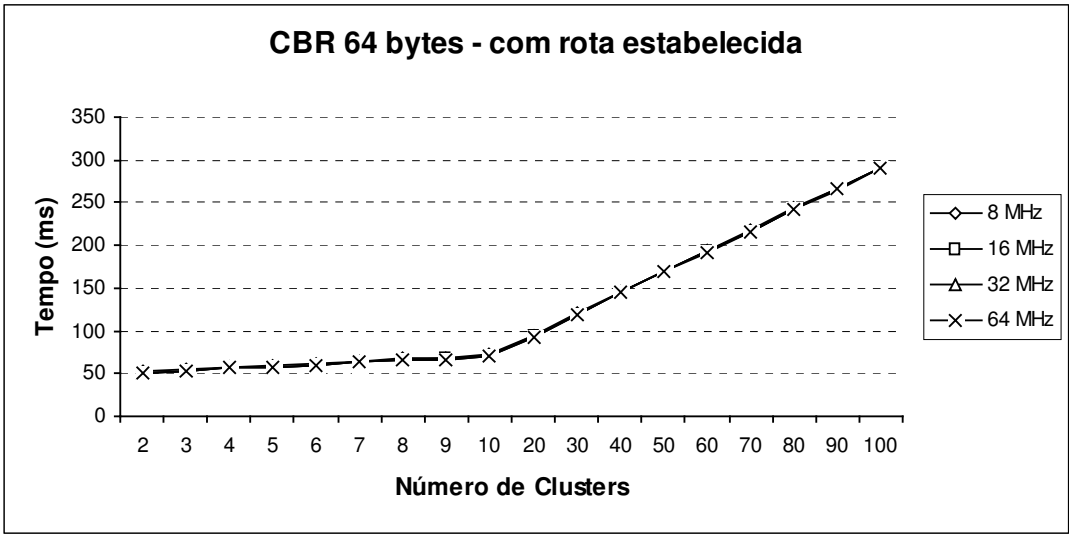

Gráfico 2: Tempo máximo para troca de uma mensagem CBR de 64 bytes com rota estabelecida

Isto se deve ao fato do poder de processamento estar relacionado a somente uma pequena parte no processo de envio da mensagem. As operações de decifrar e de cifrar os dados com as chaves de grupo e de sessão, feitas pelos *cluster heads* de origem e de destino.

A seguir são apresentados os gráficos relativos ao envio das mensagens de 128 bytes sem rota pré-estabelecida (gráfico 3) e com rota estabelecida (gráfico 4).

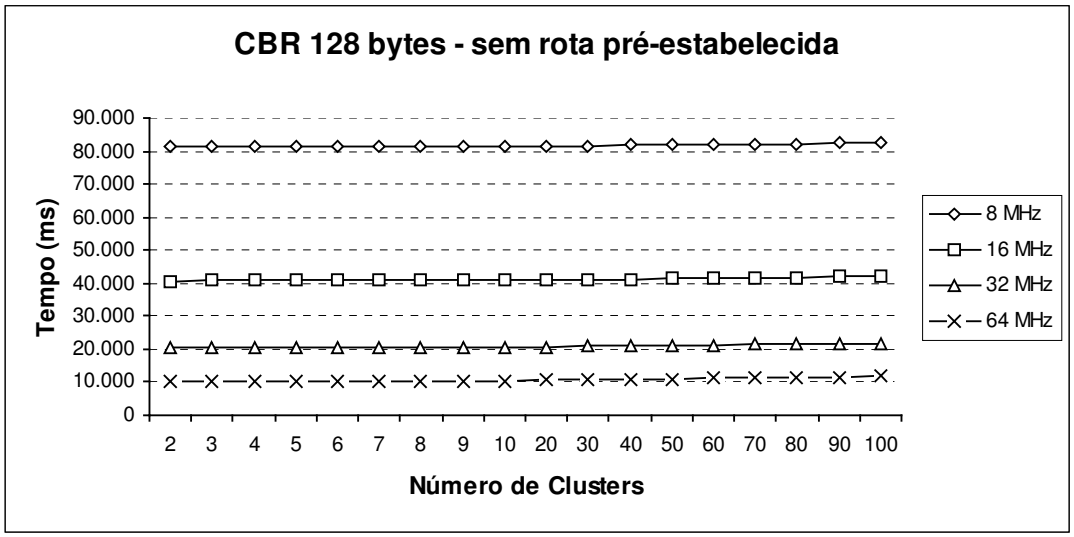

Gráfico 3: Tempo máximo para troca de uma mensagem CBR de 128 bytes sem rota pré-estabelecida

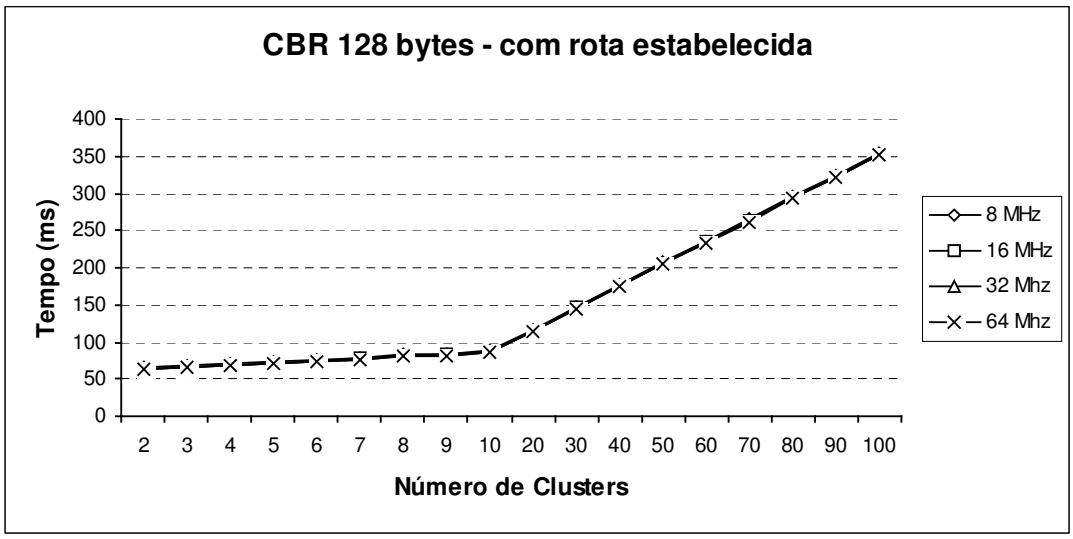

Gráfico 4: Tempo máximo para troca de uma mensagem CBR de 128 bytes com rota estabelecida

No gráfico 4 pode-se notar um acrésimo de 50 milissegundos no tempo total de envio da mensagem de 128 bytes. Esta diferença não pôde ser notada no gráfico 3. As razões são as mesmas das que já foram comentadas para os gráficos 1 e 2.

Considerando o envio de dados de 256 bytes, são apresentados o gráfico 5, sem rota pré-estabelecida e o gráfico 6, com rota estabelecida.

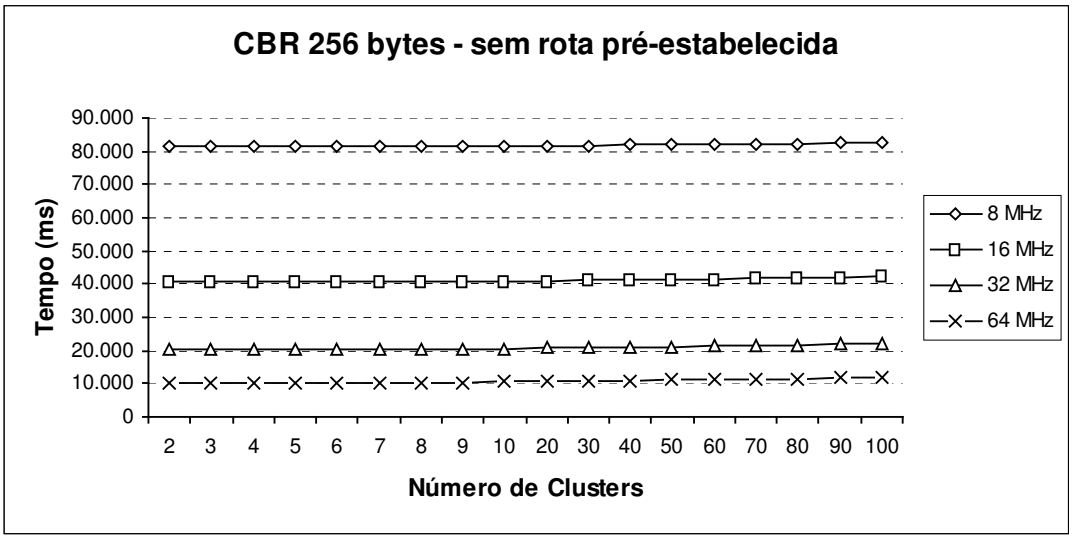

Gráfico 5: Tempo máximo para troca de uma mensagem CBR de 256 bytes sem rota pré-estabelecida

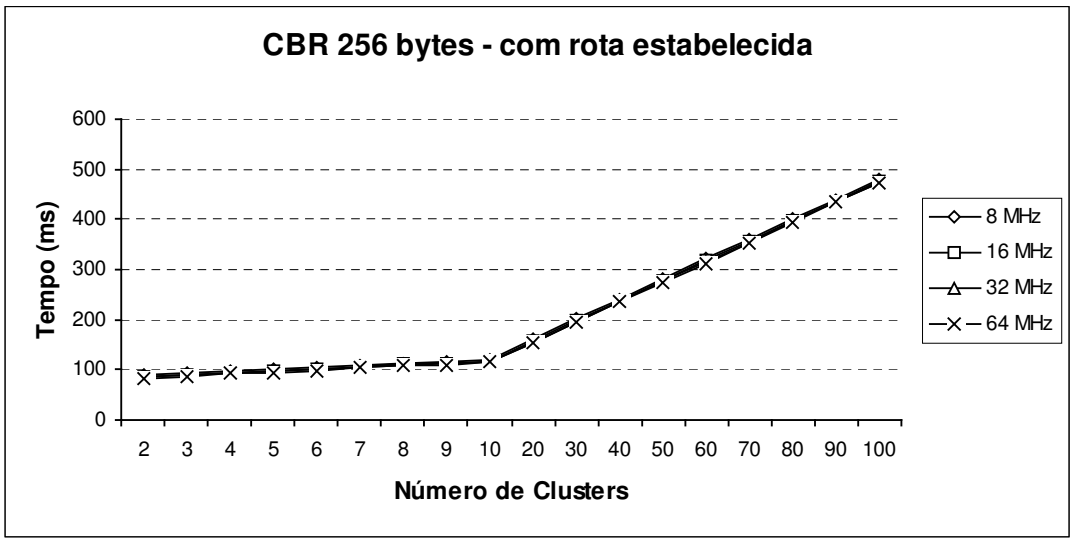

Gráfico 6: Tempo máximo para troca de uma mensagem CBR de 256 bytes com rota estabelecida

Assim como na troca de dados de tamanhos de 64 bytes e de 128 bytes, somente no envio de dados de 256 bytes com rota estabelecida é que se torna possível verificar o acréscimo no tempo de envio da mensagem. Esta percepção visual é praticamente impossível de se ter no gráfico 5 pelos mesmos motivos apresentados anteriormente.

No envio de dados de 512 bytes, o gráfico 7 e o gráfico 8 são apresentados.

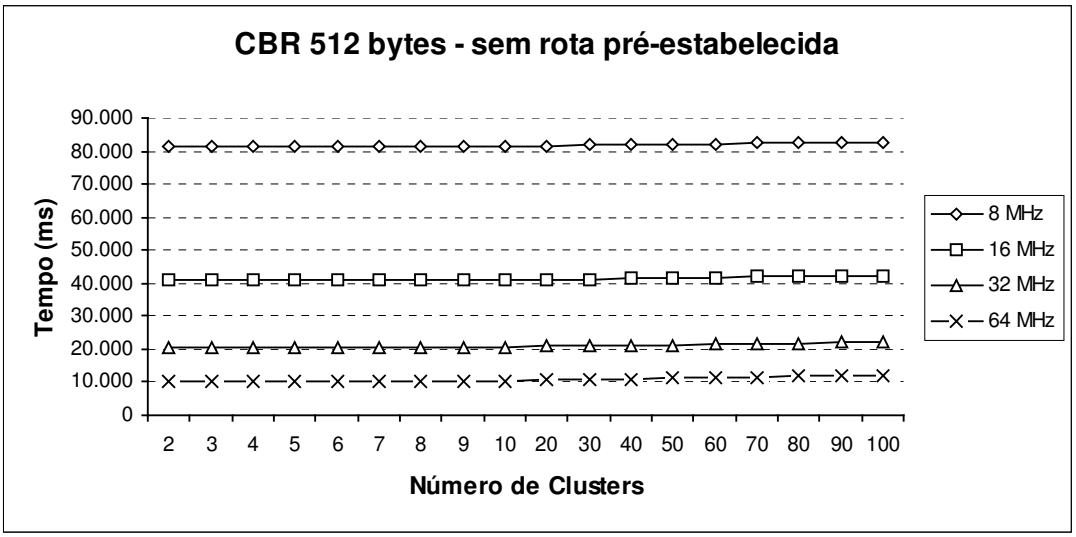

Gráfico 7: Tempo máximo para troca de uma mensagem CBR de 512 bytes sem rota pré-estabelecida

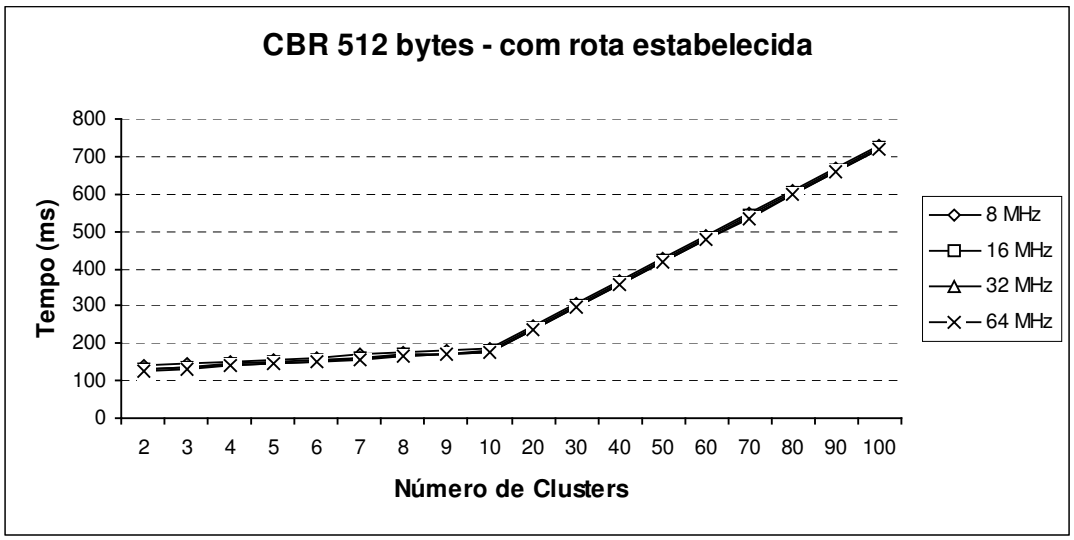

Gráfico 8: Tempo máximo para troca de uma mensagem CBR de 512 bytes com rota estabelecida

Finalmente, o envio de dados de 1024 bytes é apresentado no gráfico 9, sem uma rota pré-estabelecida e no gráfico 10, com uma rota já estabelecida.

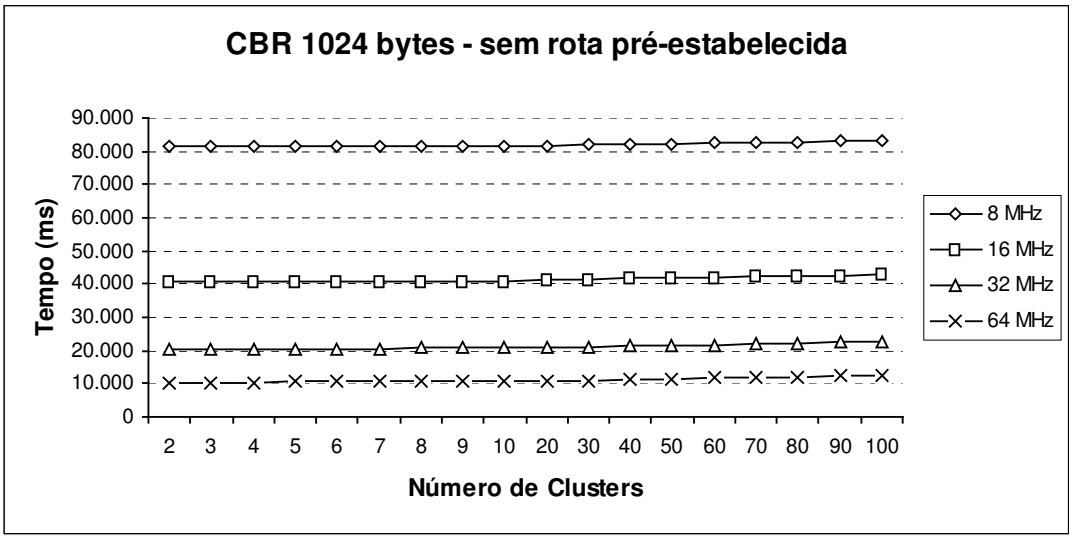

Gráfico 9: Tempo máximo para troca de uma mensagem CBR de 1024 bytes sem rota pré-estabelecida

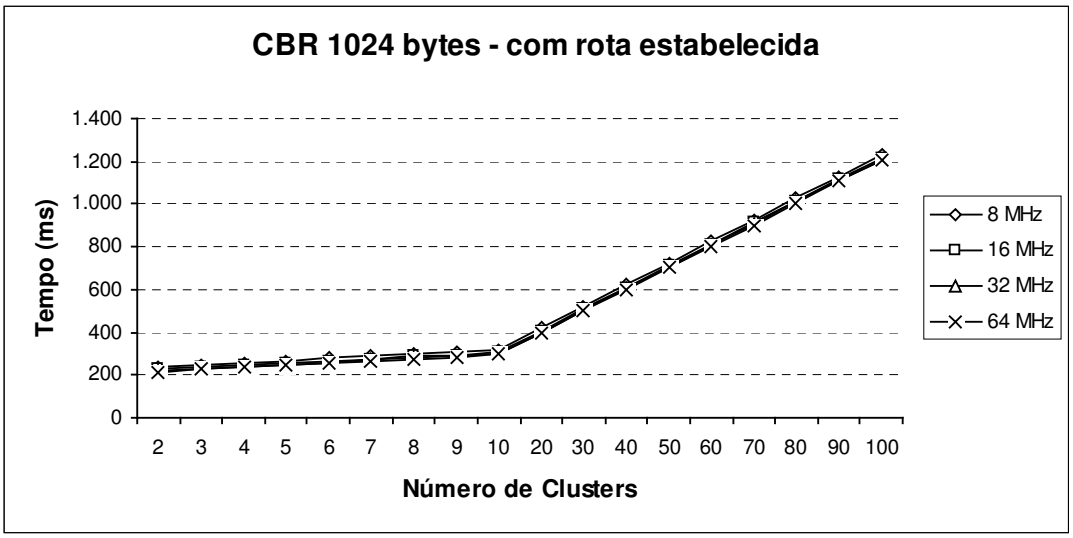

Gráfico 10: Tempo máximo para troca de uma mensagem CBR de 1024 bytes com rota estabelecida

Em geral, os gráficos que representam o envio de uma mensagem de dados sem rota pré-estabelecida não possuem alterações significativas, independentemente do tamanho das mensagens (de 64 a 1024 bytes). Além dessa característica, é possível notar que a utilização de *cluster heads* com maior capacidade de processamento torna-se um fator decisivo na obtenção de melhores resultados para o envio da mensagem.

O mesmo não se pode afirmar a partir dos gráficos que representam o envio da segunda mensagem de dados, onde a rota já está estabelecida. Nestes gráficos, o poder de processamento não tem um papel tão determinante quanto na hora de estabelecer a rota. Analisando suas curvas é possível notar um comportamento bastante similar entre elas, tendo somente um deslocamento de tempo entre cada gráfico, justamente relativo ao tamanho do pacote que se está transmitindo.

Para uma melhor visualização e entendimento dos valores obtidos pelas simulações, os gráficos a seguir apresentam os mesmos valores só que de uma ótica diferente. São separados por característica de *cluster head*, fazendo o cruzamento entre os tamanhos de mensagens de dados que são transferidas. Da mesma forma que anteriormente, é apresentada a mesma abordagem de tipos de gráficos, sem rota pré-estabelecida e com a rota já estabelecida.

A seguir, são apresentados nos gráficos 11 e 12 os resultados obtidos com o envio de mensagens de dados utilizando *cluster heads* com poder de processamento de 8 MHz, respectivamente, sem e com rota pré-estabelecida.

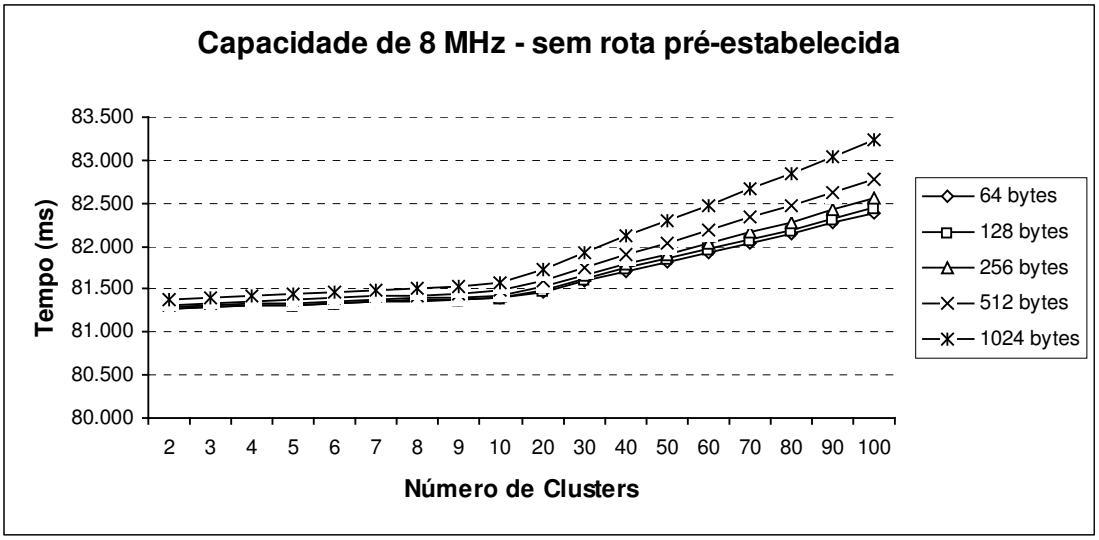

Gráfico 11: Envio de mensagem utilizando *cluster heads* de 8Mhz sem rota pré-estabelecida

No gráfico 11 é possível notar que as curvas que representam as mensagens de dados de tamanhos maiores, apresentam uma inclinação maior. Desta forma, podemos afirmar que a inclinação da curva (aumento do tempo total de envio da mensagem) está diretamente ligada ao tamanho da mensagem de dados que se quer enviar. É importante ressaltar que quando eram utilizadas escalas maiores para a representação destas informações (gráficos anteriores), esta inclinação era praticamente imperceptível.

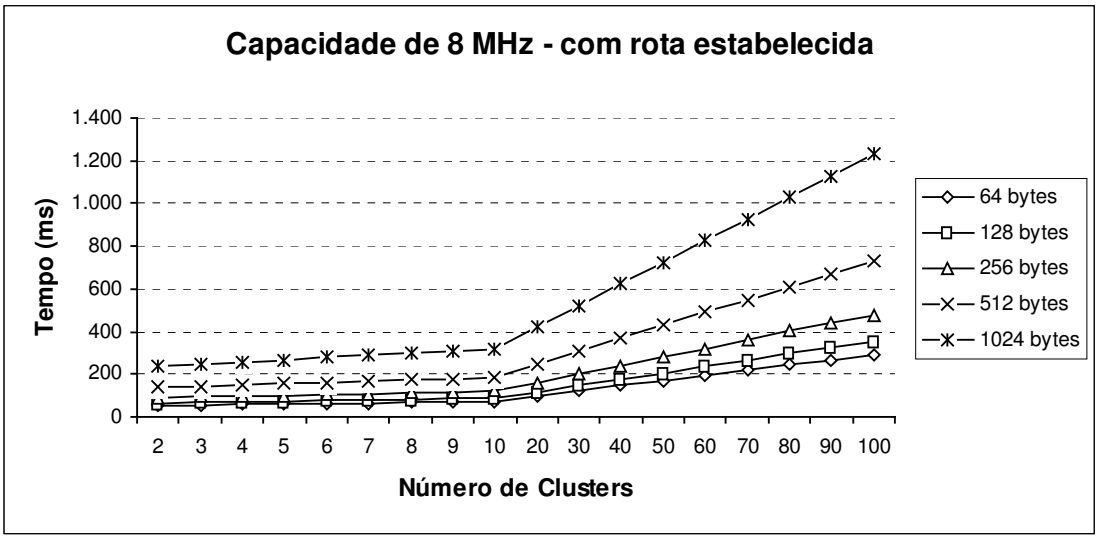

Gráfico 12: Envio de mensagem utilizando *cluster heads* de 8Mhz com rota pré-estabelecida

A inclinação verificada no gráfico 11 fica mais evidente no gráfico 12, onde a relação do tempo representado no gráfico e o aumento gerado pelo tamanho da mensagem são mais relevantes.

A seguir são apresentados os gráficos 13 e 14 sob a ótica do uso de *cluster heads* com 16 MHz de processamento.

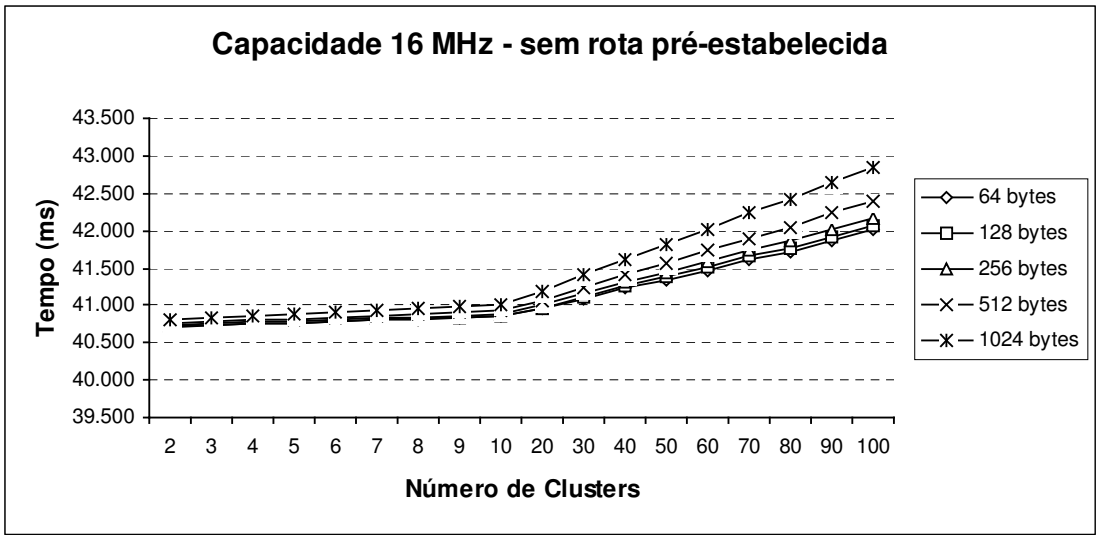

Gráfico 13: Envio de mensagem utilizando *cluster heads* de 16 Mhz sem rota pré-estabelecida

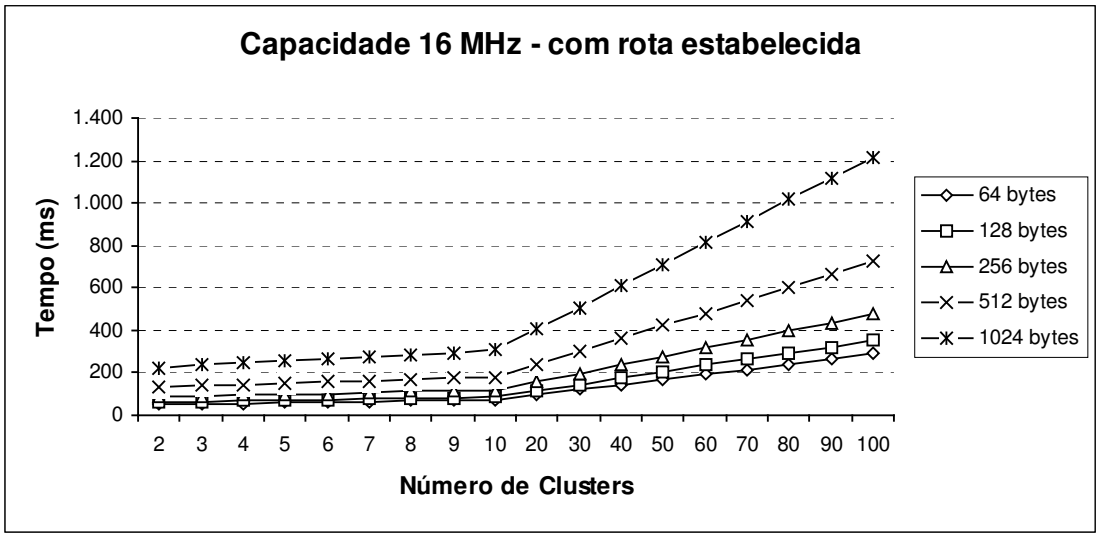

Gráfico 14 : Envio de mensagem utilizando *cluster heads* de 16 Mhz com rota pré-estabelecida

Ambos os gráficos possuem comportamento semelhante. Principalmente se for considerado o envio de dados com rota já estabelecida. No caso do estabelecimento da rota, há uma redução no tempo em quase metade, devido ao que já foi exposto anteriormente. Da mesma forma, no gráfico 13 pode-se observar uma inclinação na curva que não ficava tão evidente nos gráficos que agrupavam os tempos por tamanho do pacote de dados transmitidos.

A seguir os gráficos 15 e 16 apresentam os resultados obtidos através do uso de *cluster heads* de 32 MHz, sem e com rota pré-estabelecida, respectivamente.

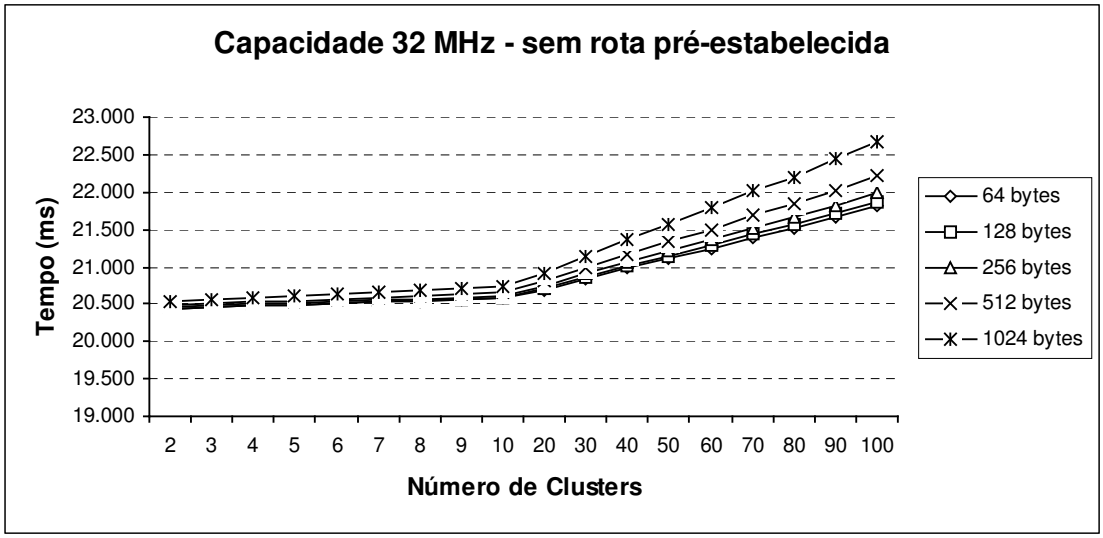

Gráfico 15: Envio de mensagem utilizando *cluster heads* de 32 Mhz sem rota pré-estabelecida

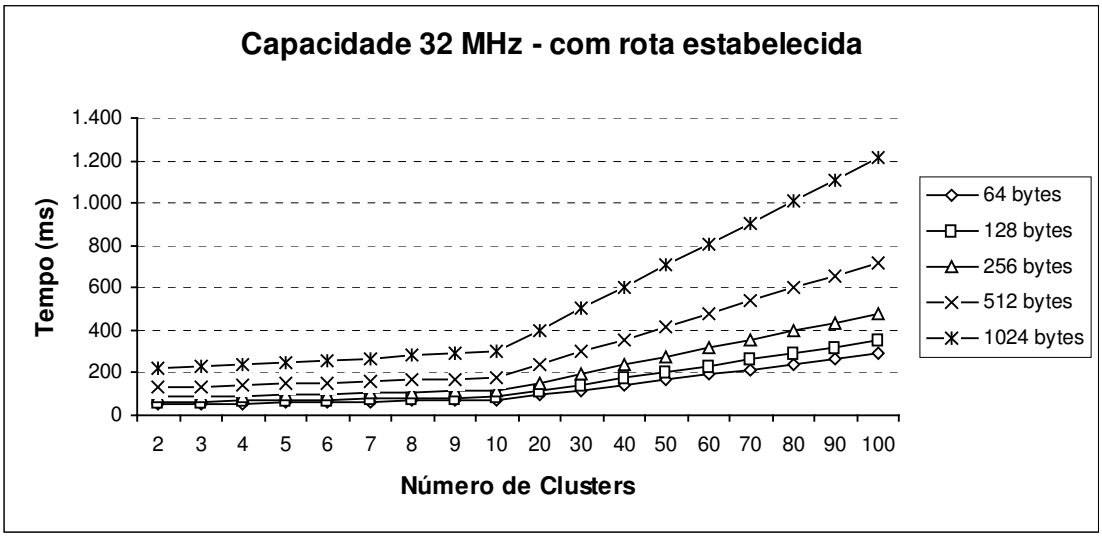

Gráfico 16: Envio de mensagem utilizando *cluster heads* de 32 Mhz com rota pré-estabelecida

As mesmas observações feitas em relação aos gráficos 12 e 14 se aplicam ao gráfico 16, pois possuem valores semelhantes, e onde se nota que o poder de processamento não afeta o desempenho do envio das diferentes mensagens de dados. O gráfico 15 tem comportamento semelhante ao gráfico 13, só que com os tempos reduzidos pela metade.

Em seguida são apresentados os resultados obtidos pelo uso de *cluster heads* de 64 Mhz, através dos gráficos 17 e 18, relativos ao envio de mensagem de dados CBR sem e com rota pré-estabelecida, respectivamente.

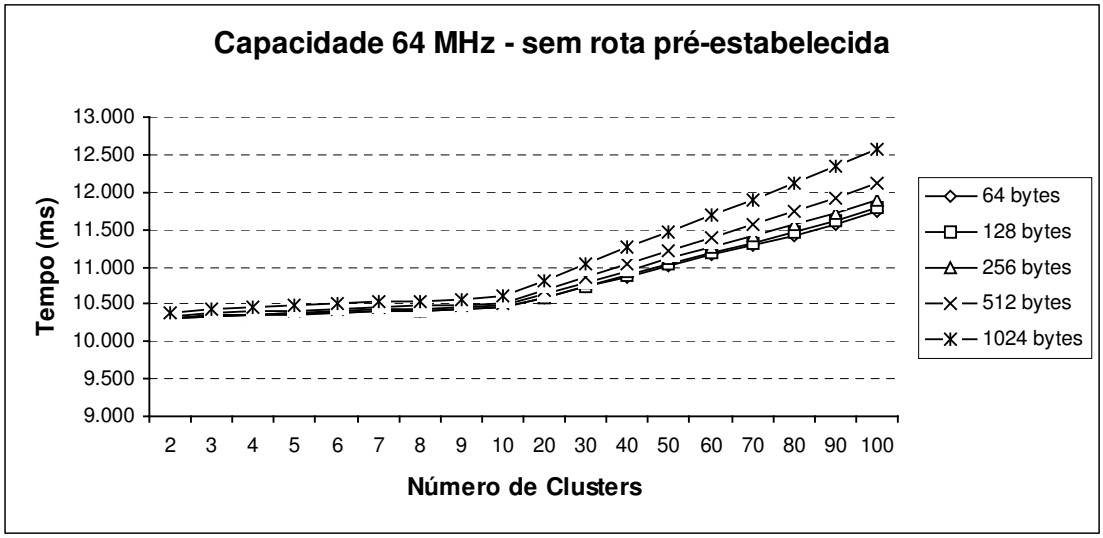

Gráfico 17: Envio de mensagem utilizando *cluster heads* de 64 Mhz sem rota pré-estabelecida

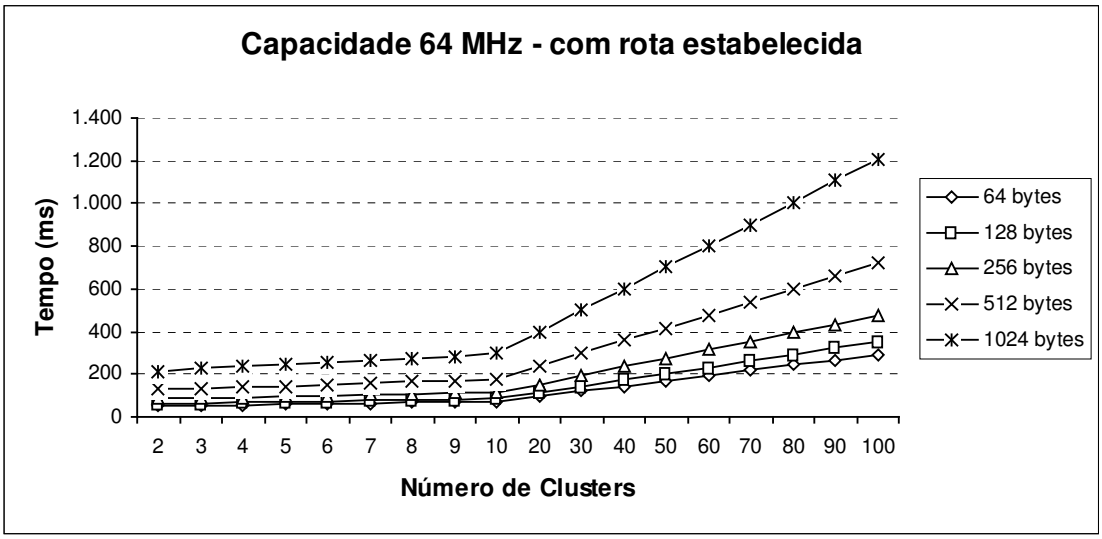

Gráfico 18: Envio de mensagem utilizando *cluster heads* de 64 Mhz com rota pré-estabelecida

Finalmente, analisando os dados sob a ótica do uso de *cluster heads* de 64 MHz, pode-se concluir que no estabelecimento da rota, quando se dobra o poder de processamento, o tempo de envio da mensagem é reduzido pela metade. Esta vantagem, porém, não é verificada no envio da segunda mensagem, quando a rota já está estabelecida. O que determina a inclinação das curvas é o tamanho das mensagens e conseqüentemente o tempo de propagação das mensagens no espaço.

6. CONCLUSÕES E TRABALHOS FUTUROS

O presente trabalho apresenta um esquema de distribuição e proteção de chaves simétricas e propõe um mecanismo para interconectar de maneira transparente dois *clusters*. Para a interconexão de *clusters*, foi adaptado o protocolo AODV para o ambiente desejado.

Verificando o resultado das simulações, nota-se um desempenho uniforme no processo de descobrimento de rotas para a interconexão de *clusters*, mesmo quando o número de *clusters* é aumentado consideravelmente. Também pode ser constatado que o uso de *cluster heads* com maior poder de processamento é extremamente importante para reduzir o tempo no estabelecimento de rotas e envio da primeira mensagem de dados. O mesmo já não pode ser concluído depois que a rota está estabelecida, pois observa-se um comportamento semelhante nos tempos de envio de mensagem, quando se utiliza *cluster heads* mais potentes.

As simulações ainda mostram o desempenho da interconexão de *clusters* levando em consideração o tamanho do pacote de dados a ser enviado. Pelos resultados obtidos, observa-se que quanto maior o tamanho do pacote, mais tempo é gasto para transmiti-lo entre os componentes que participam da rota de comunicação. Este tempo está relacionado com a propagação da mensagem no espaço e com as operações de cifrar e decifrar a mensagem de dados com as chaves utilizadas em cada trecho da comunicação. É importante ressaltar que o aumento no tamanho dos pacotes não apresenta um acréscimo possível de ser notado visualmente no envio da primeira mensagem de dados, pois o tempo envolvido para o estabelecimento da rota é na ordem de milhares de vezes maior.

Analisando o protocolo de distribuição de chaves de grupo, pode-se concluir que o mesmo cumpre a finalidade ao qual foi desenvolvido, isto é, permite a distribuição de chaves aos sensores, e permite a criação do ambiente de *clusters*. Pelo foco das simulações, não houve uma análise aprofundada para poder determinar o desempenho do mesmo. Inclusive a construção do protocolo de distribuição de chaves é motivada justamente pelo modelo proposto para a interconexão dos *clusters* e pela maneira como as chaves de sessão são utilizadas.

Algumas desvantagens desta abordagem devem ser citadas, tais como o modo como as mensagens trafegam, que podem sobrecarregar o *cluster head*, tornando-o um gargalo na comunicação *extra cluster*. No esquema de distribuição de chaves, como as requisições de chave de grupo não tem a sua autenticidade verificada pelos sensores intermediários, pode-se criar um DoS no *cluster head*, caso um nó intruso fique enviando este tipo de solicitação.

Outra desvantagem que pode ser destacada é a maneira como são tratadas as autenticações de mensagens entre *cluster heads* intermediários, podendo haver um retardo na identificação de uma mensagem anômala. Mesmo assim acredita-se que os benefícios obtidos neste trabalho sejam maiores que as eventuais desvantagens. O principal atrativo desta abordagem é ter um desempenho constante na troca de mensagens entre dois *clusters* quaisquer, mesmo com o aumento do número de sensores.

Como o foco principal do trabalho foi destinado à interconexão de *clusters*, e conseqüentemente pelas simulações terem sido feitas para este fim, a avaliação do protocolo de distribuição de chaves mais aprofundada fica como sugestão para trabalhos futuros. A sugestão para trabalhos futuros se extende ao o desafio de alterar o protocolo implementado para o seu funcionamento considerar a mobilidade dos sensores. Também apresentar uma solução para as extensões de assinatura dupla que estão previstas no trabalho inicial de Zapata e Asokan (2002) e que não foram implementadas na solução apresentada neste trabalho. A extensão deste trabalho pode ainda considerar a questão do consumo de energia, visto que os sensores são dispositivos que utializam baterias. E uma das principais sugestões é que este ambiente possa ser comprovado na prática, implementando a solução em sensores e confrontando com os resultados obtidos através das simulações.

REFERÊNCIAS BIBLIOGRÁFICAS

ANJUN, F. e SARKAR, S. **Security in Sensor Networks**. In: SHOREY, R. et al. Mobile, wireless, and sensor networks: technology, applications, and future directions. New Jersey: John Wiley & Sons, Inc/IEEE Press, 2006. pp. 283-307.

ARAZI, B., ELHANANY, I., ARAZI, O. e QI, H. **Revisiting Public-Key Cryptography for Wireless Sensor Network**. IEEE Computer, Vol. 38, Num. 11. Novembro/2005.

BASAGNI, S., HERRÍN, K., BRUSCHI, D. e ROSTI, E. **Secure Pebblenets**. In Proc. of the ACM Int. Symp. On Mobile Ad Hoc Networking and computing, ACM Press, New York. 2001. pp 156-163.

BECHLER, M., HOF, H. J., KRAFT, D., PÄHLKE, F. e WOLF, L. **A cluster-based security architecture for ad hoc network s**. IEEE INFOCOM. 2004.

BLAß, E.-O. e ZITTERBART, M. **Efficient Implementation of Elliptic Curve Cryptography for Wireless Sensor Networks**. Institute of Telematics, University of Karlsruhe. http://doc.tm.uka.de/tr/TM-2005-1.pdf, Março/2005.

CERTICOM. **Certicom's Bulletin of Security and Cryptography**: Code & Cipher, Certicom's.Bulletin of Security and Cryptography. Disponível em: <www.certicom.com>. Acesso em: 12 Novembro de 2006.

CHAN, H., PERRIG, A. e SONG, D. **Randon key predistribution schemes for sensor networks**. In IEEE Symposium on Security and Privacy. IEEE Computer Society, 2003.

DU, W., WANG, R. e NING, P. **An Efficient Scheme for Authenticating Public Keys in Sensor Networks**". MobiHoc'05. Illinois. USA. Maio/2005.

ESCHENAUER, L. e GLIGOR, V. D. **A key-management scheme for distributed sensor networks**. In Proc. of the 9th ACM Conf. On Computer and Communications Security, pp. 41-47. Novembro/2002.

FOKINE, K. **Key management in ad hoc networks**. Master Thesis. http://www.ep.liu.se/exjobb/isy/2002/3322/. Setembro/2002.

GANESAN, P., VENUGOPALAN, R. PEDDABACHAGARI, P., DEAN, A., MUELLER, F. e SICHITIU, M. **Analyzing and Modeling Encryption Overhead for Sensor Network Nodes**. Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications. 2003. pp. 151-159.

GURA, N., PATEL, A., WANDER, A., EBERLE, H. e CHANG-SHANTZ, S. **Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs**. CHES'2004. Workshop on Cryptographic Hardware and Embedded Systems. Springer-Verlag. Agosto/2004.

HU, F. e SHARMA, N. K. **Security considerations in ad hoc sensor networks**. Ad Hoc Networks 3. 2005. pp. 69-89.

HUBAUX, J., BUTTYAN, L., e CAPKUN, S. **The quest for security in mobile ad hoc networks**. In Proc. ACM Symp. On Mobile Ad Hoc Networking and Computing (MobiHOC). 2001.

KAMAL, A. B. M. **Adaptive Secure Routing in Ad Hoc Mobile Network**. Master of Science Thesis. Royal Institute of Technology. Sweden. Novembro/2004.

LENSTRA, A. K. e VERHEUL, E. R. **Selecting Cryptographic Key Sizes**. Journal of Cryptology: the Journal of the International Association for Cryptologic Research. 2001. 14(4): 255-293.

LIU, D. e NING, P. Establishing pairwise keys in distributed sensor networks. In 10th ACM Conference on Computer and Communications Security. ACM Press, 2003. pp.52-61

MALAN, D. J., WELSH, M. e SMITH, M.D. **A Public-Key Infrastructure for Key Distribuition in TinyOS Based Elliptic Curve Cryptography**. First IEEE International Conference on Sensor and Ad Hoc Communications and Networks. 2004.

MENEZES, A., Van OORSCHOT, P., VANSTONE, S. **Handbook of Applied Cryptography**". Boca Raton, FL. CRC Press, 1997.

MENEZES, A. G., WESTPHALL, C. B. **Security Approaches for Cluster Interconnection in a Wireless Sensor Network.** In: 9th Asia-Pacific Network Operations and Management Symposium, APNOMS 2006 Busan, Korea, September 27-29, 2006. pp. 542-545.

MENEZES, A. G., WESTPHALL, C. B. **Interconectando Clusters com Transparência em Rede de Sensores Sem Fio Segura.** In: VI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, SBSeg 2006. Agosto/Setembro de 2006.

OKABE, N., SAKANE, S., MIYAZAWA, K., KAMADA, K., INOUE, A. e ISHIYAMA, M. **Security Architecture for Networks using Ipsec and KINK**. SAINT'2005. International Symposium on Applications and the Internet. IEEE. Computer Society. 2005.

OLIVEIRA, L.B., WONG, H. C., BERN, M., DAHAB, R., LOUREIRO, A. A. F. **SecLEACH – A Random Key Distribution Solution for Securing Clustered Sensor**

Networks. Fifth IEEE International Symposium on Network Computing and Applications, 2006.

PERKINS, C. e BELDING-ROYER, E. **Ad hoc on-demand distance vector (AODV) routing**. IETF Request for Comments, RFC 3561. Julho/2003.

PERRIG, A., SZEWCZYK, R., WEN, V., CULLER, D. E., e TYGAR, J. D. **SPINS: security protocols for sensor networks**. In Proc. of the Seventh Annual Int. Conf. on Mobile Computing and Networking, 2001. pp 189-199.

SCHNEIER, B. **Applied Cryptography: Protocols, Algorithms and Source code in C**. New York. John Wiley & Sons, Inc. 1996.

SHOUP, V. **Practical threshold signatures**. EUROCRYPT´00. 2000. pp. 207–220.

STALLINGS, W. **Cryptography and Network Security: Principles and Practice**. Second edition. New Jersey. Prentice Hall, Inc. 1999.

ZAPATA, M. G., e ASOKAN, N. **Securing ad hoc routing protocols**. Proc. ACM Workshop on Wireless Security (WiSe), ACM Press, 2002. pp. 1-10.

ZHOU, L. e HAAS, Z. J. **Securing ad hoc networks**. IEEE Networkd Magazine, 1999. 13(6), pp 24-30. Novembro/Dezembro.

ZHU, S. SETIA, S. e JAJODIA, S. **LEAP: efficient security mechanisms for large-scale distributed sensor networks**. In Proceedings of the 10th ACM Conference on Computer and Communications Security. ACM Press, 2003. pp. 62-72.

ANEXO I

Conteúdo do arquivo **002_clusters.tr**, que foi originado pelo descobrimento de rotas entre dois clusters. O poder de processamento do cluster head é de 8MHz e o tamanho do pacote de dados é de 512 bytes

```
s 0.000000000 _0_ AGT  --- 0 cbr 512 [0 0 0 0] ------- [1:0 2058:0 32 0] [0] 0 0
r 0.000000000 _0_ RTR  --- 0 cbr 512 [0 0 0 0] ------- [1:0 2058:0 32 0] [0] 0 0
f 0.004462000 _0_ RTR  --- 0 AODV 80 [0 0 0 0] ------- [1:255 -1:255 120 0] [0x2 1 1 [2058 0]
[1 4]] (REQUEST_INTRACLUSTER)
r 0.006078033 _1_ RTR  --- 0 AODV 80 [0 ffffffff 0 800] ------- [1:255 -1:255 120 0] [0x2 1 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 0.006220080 _1_ RTR  --- 0 AODV 80 [0 ffffffff 0 800] ------- [2:255 -1:255 119 0] [0x2 2 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.007676113 _0_ RTR  --- 0 AODV 80 [0 ffffffff 1 800] ------- [2:255 -1:255 119 0] [0x2 2 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.007676113 _2_ RTR  --- 0 AODV 80 [0 ffffffff 1 800] ------- [2:255 -1:255 119 0] [0x2 2 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 0.015667668 _2_ RTR  --- 0 AODV 80 [0 ffffffff 1 800] ------- [3:255 -1:255 118 0] [0x2 3 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.016983701 _1_ RTR  --- 0 AODV 80 [0 ffffffff 2 800] ------- [3:255 -1:255 118 0] [0x2 3 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.016983701 _3_ RTR  --- 0 AODV 80 [0 ffffffff 2 800] ------- [3:255 -1:255 118 0] [0x2 3 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 0.019329719 _3_ RTR  --- 0 AODV 80 [0 ffffffff 2 800] ------- [4:255 -1:255 117 0] [0x2 4 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.020845752 _2_ RTR  --- 0 AODV 80 [0 ffffffff 3 800] ------- [4:255 -1:255 117 0] [0x2 4 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.020845752 _4_ RTR  --- 0 AODV 80 [0 ffffffff 3 800] ------- [4:255 -1:255 117 0] [0x2 4 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 0.029208987 _4_ RTR  --- 0 AODV 80 [0 ffffffff 3 800] ------- [5:255 -1:255 116 0] [0x2 5 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.030405021 _3_ RTR  --- 0 AODV 80 [0 ffffffff 4 800] ------- [5:255 -1:255 116 0] [0x2 5 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.030405021 _5_ RTR  --- 0 AODV 80 [0 ffffffff 4 800] ------- [5:255 -1:255 116 0] [0x2 5 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 0.036805697 _5_ RTR  --- 0 AODV 80 [0 ffffffff 4 800] ------- [6:255 -1:255 115 0] [0x2 6 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.038321731 _4_ RTR  --- 0 AODV 80 [0 ffffffff 5 800] ------- [6:255 -1:255 115 0] [0x2 6 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.038321731 _6_ RTR  --- 0 AODV 80 [0 ffffffff 5 800] ------- [6:255 -1:255 115 0] [0x2 6 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 0.042558298 _6_ RTR  --- 0 AODV 80 [0 ffffffff 5 800] ------- [7:255 -1:255 114 0] [0x2 7 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.044194331 _5_ RTR  --- 0 AODV 80 [0 ffffffff 6 800] ------- [7:255 -1:255 114 0] [0x2 7 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.044194331 _7_ RTR  --- 0 AODV 80 [0 ffffffff 6 800] ------- [7:255 -1:255 114 0] [0x2 7 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 0.052264617 _7_ RTR  --- 0 AODV 80 [0 ffffffff 6 800] ------- [8:255 -1:255 113 0] [0x2 8 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.053620650 _6_ RTR  --- 0 AODV 80 [0 ffffffff 7 800] ------- [8:255 -1:255 113 0] [0x2 8 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.053620650 _8_ RTR  --- 0 AODV 80 [0 ffffffff 7 800] ------- [8:255 -1:255 113 0] [0x2 8 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 0.057707507 _8_ RTR  --- 0 AODV 80 [0 ffffffff 7 800] ------- [9:255 -1:255 112 0] [0x2 9 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.059103540 _7_ RTR  --- 0 AODV 80 [0 ffffffff 8 800] ------- [9:255 -1:255 112 0] [0x2 9 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.059103540 _9_ RTR  --- 0 AODV 80 [0 ffffffff 8 800] ------- [9:255 -1:255 112 0] [0x2 9 1
[2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 0.066026609 _9_ RTR  --- 0 AODV 80 [0 ffffffff 8 800] ------- [10:255 -1:255 111 0] [0x2 10
1 [2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.067762642 _8_ RTR  --- 0 AODV 80 [0 ffffffff 9 800] ------- [10:255 -1:255 111 0] [0x2 10
1 [2058 0] [1 4]] (REQUEST_INTRACLUSTER)
r 0.067762642 _20_ RTR  --- 0 AODV 80 [0 ffffffff 9 800] ------- [10:255 -1:255 111 0] [0x2 10
1 [2058 0] [1 4]] (REQUEST_INTRACLUSTER)
f 7.404090642 _20_ RTR  --- 0 AODV 156 [0 0 0 0] ------- [0:255 -1:255 110 0] [0x12 11 1 [2058
0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406394675 _9_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
```

```
r 7.406394709 _8_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406394742 _7_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406394775 _6_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406394809 _5_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406394842 _4_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406394875 _3_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406394909 _2_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406394942 _1_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406394975 _0_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
r 7.406395309 _21_ RTR  --- 0 AODV 156 [0 ffffffff 14 800] ------- [0:255 -1:255 110 0] [0x12
11 1 [2058 0] [1 4]] (REQUEST_EXTRACLUSTER)
f 32.004653309 _21_ RTR  --- 0 AODV 68 [0 0 0 0] ------- [2048:255 -1:255 2 0] [0x6 0 [1
2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753342 _10_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753375 _11_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753409 _12_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753442 _13_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753475 _14_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753509 _15_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753542 _16_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753575 _17_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753609 _18_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753642 _19_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
r 32.005753975 _20_ RTR  --- 0 AODV 68 [0 ffffffff 15 800] ------- [2048:255 -1:255 2 0] [0x6
0 [1 2058]] (ROUTE_REQUEST_INITIATOR)
f 32.014669642 _19_ RTR  --- 0 AODV 80 [0 0 0 0] ------- [2058:255 -1:255 120 0] [0x2 1 1
[2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.016185675 _18_ RTR  --- 0 AODV 80 [0 ffffffff 13 800] ------- [2058:255 -1:255 120 0]
[0x2 1 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.020240866 _18_ RTR  --- 0 AODV 80 [0 ffffffff 13 800] ------- [2057:255 -1:255 119 0]
[0x2 2 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.021676899 _17_ RTR  --- 0 AODV 80 [0 ffffffff 12 800] ------- [2057:255 -1:255 119 0]
[0x2 2 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.021676899 _19_ RTR  --- 0 AODV 80 [0 ffffffff 12 800] ------- [2057:255 -1:255 119 0]
[0x2 2 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.023550552 _17_ RTR  --- 0 AODV 80 [0 ffffffff 12 800] ------- [2056:255 -1:255 118 0]
[0x2 3 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.025226585 _16_ RTR  --- 0 AODV 80 [0 ffffffff 11 800] ------- [2056:255 -1:255 118 0]
[0x2 3 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.025226585 _18_ RTR  --- 0 AODV 80 [0 ffffffff 11 800] ------- [2056:255 -1:255 118 0]
[0x2 3 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.034140063 _16_ RTR  --- 0 AODV 80 [0 ffffffff 11 800] ------- [2055:255 -1:255 117 0]
[0x2 4 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.035876097 _15_ RTR  --- 0 AODV 80 [0 ffffffff 10 800] ------- [2055:255 -1:255 117 0]
[0x2 4 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.035876097 _17_ RTR  --- 0 AODV 80 [0 ffffffff 10 800] ------- [2055:255 -1:255 117 0]
[0x2 4 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.035984180 _15_ RTR  --- 0 AODV 80 [0 ffffffff 10 800] ------- [2054:255 -1:255 116 0]
[0x2 5 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.037480213 _14_ RTR  --- 0 AODV 80 [0 ffffffff f 800] ------- [2054:255 -1:255 116 0] [0x2
5 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.037480213 _16_ RTR  --- 0 AODV 80 [0 ffffffff f 800] ------- [2054:255 -1:255 116 0] [0x2
5 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.038711248 _14_ RTR  --- 0 AODV 80 [0 ffffffff f 800] ------- [2053:255 -1:255 115 0] [0x2
6 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.040047282 _13_ RTR  --- 0 AODV 80 [0 ffffffff e 800] ------- [2053:255 -1:255 115 0] [0x2
6 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
```

```
r 32.040047282 _15_ RTR  --- 0 AODV 80 [0 ffffffff e 800] ------- [2053:255 -1:255 115 0] [0x2
6 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.047034014 _13_ RTR  --- 0 AODV 80 [0 ffffffff e 800] ------- [2052:255 -1:255 114 0] [0x2
7 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.048190047 _12_ RTR  --- 0 AODV 80 [0 ffffffff d 800] ------- [2052:255 -1:255 114 0] [0x2
7 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.048190047 _14_ RTR  --- 0 AODV 80 [0 ffffffff d 800] ------- [2052:255 -1:255 114 0] [0x2
7 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.050808289 _12_ RTR  --- 0 AODV 80 [0 ffffffff d 800] ------- [2051:255 -1:255 113 0] [0x2
8 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.052124322 _11_ RTR  --- 0 AODV 80 [0 ffffffff c 800] ------- [2051:255 -1:255 113 0] [0x2
8 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.052124322 _13_ RTR  --- 0 AODV 80 [0 ffffffff c 800] ------- [2051:255 -1:255 113 0] [0x2
8 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.052399897 _11_ RTR  --- 0 AODV 80 [0 ffffffff c 800] ------- [2050:255 -1:255 112 0] [0x2
9 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.053855930 _10_ RTR  --- 0 AODV 80 [0 ffffffff b 800] ------- [2050:255 -1:255 112 0] [0x2
9 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.053855930 _12_ RTR  --- 0 AODV 80 [0 ffffffff b 800] ------- [2050:255 -1:255 112 0] [0x2
9 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.061705801 _10_ RTR  --- 0 AODV 80 [0 ffffffff b 800] ------- [2049:255 -1:255 111 0] [0x2
10 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.063161835 _21_ RTR  --- 0 AODV 80 [0 ffffffff a 800] ------- [2049:255 -1:255 111 0] [0x2
10 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
r 32.063161835 _11_ RTR  --- 0 AODV 80 [0 ffffffff a 800] ------- [2049:255 -1:255 111 0] [0x2
10 1 [2048 0] [2058 4]] (REQUEST_INTRACLUSTER)
f 32.063161835 _21_ RTR  --- 0 AODV 76 [0 0 0 0] ------- [2048:255 2058:255 120 2049] [0x4 1
[2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.067812068 _10_ RTR  --- 0 AODV 76 [13a a 15 800] ------- [2048:255 2058:255 120 2049]
[0x4 1 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 32.072266068 _10_ RTR  --- 0 AODV 76 [13a a 15 800] ------- [2048:255 2058:255 119 2050]
[0x4 2 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.077301301 _11_ RTR  --- 0 AODV 76 [13a b a 800] ------- [2048:255 2058:255 119 2050] [0x4
2 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 32.081755301 _11_ RTR  --- 0 AODV 76 [13a b a 800] ------- [2048:255 2058:255 118 2051] [0x4
3 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.086250535 _12_ RTR  --- 0 AODV 76 [13a c b 800] ------- [2048:255 2058:255 118 2051] [0x4
3 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 32.090704535 _12_ RTR  --- 0 AODV 76 [13a c b 800] ------- [2048:255 2058:255 117 2052] [0x4
4 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.095579768 _13_ RTR  --- 0 AODV 76 [13a d c 800] ------- [2048:255 2058:255 117 2052] [0x4
4 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 32.100033768 _13_ RTR  --- 0 AODV 76 [13a d c 800] ------- [2048:255 2058:255 116 2053] [0x4
5 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.105629001 _14_ RTR  --- 0 AODV 76 [13a e d 800] ------- [2048:255 2058:255 116 2053] [0x4
5 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 32.110083001 _14_ RTR  --- 0 AODV 76 [13a e d 800] ------- [2048:255 2058:255 115 2054] [0x4
6 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.115378235 _15_ RTR  --- 0 AODV 76 [13a f e 800] ------- [2048:255 2058:255 115 2054] [0x4
6 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 32.119832235 _15_ RTR  --- 0 AODV 76 [13a f e 800] ------- [2048:255 2058:255 114 2055] [0x4
7 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.124827468 _16_ RTR  --- 0 AODV 76 [13a 10 f 800] ------- [2048:255 2058:255 114 2055]
[0x4 7 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 32.129281468 _16_ RTR  --- 0 AODV 76 [13a 10 f 800] ------- [2048:255 2058:255 113 2056]
[0x4 8 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.134116701 _17_ RTR  --- 0 AODV 76 [13a 11 10 800] ------- [2048:255 2058:255 113 2056]
[0x4 8 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 32.138570701 _17_ RTR  --- 0 AODV 76 [13a 11 10 800] ------- [2048:255 2058:255 112 2057]
[0x4 9 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.143205935 _18_ RTR  --- 0 AODV 76 [13a 12 11 800] ------- [2048:255 2058:255 112 2057]
[0x4 9 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 32.147659935 _18_ RTR  --- 0 AODV 76 [13a 12 11 800] ------- [2048:255 2058:255 111 2058]
[0x4 10 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
r 32.152935168 _19_ RTR  --- 0 AODV 76 [13a 13 12 800] ------- [2048:255 2058:255 111 2058]
[0x4 10 [2048 6] 100.000000] (REPLY_INTRACLUSTER)
f 39.359493835 _21_ RTR  --- 0 AODV 152 [0 0 0 0] ------- [2048:255 1:255 110 0] [0x14 11
[2058 4] 100.000000] (REPLY_EXTRACLUSTER)
r 39.364561501 _20_ RTR  --- 0 AODV 152 [13a 14 15 800] ------- [2048:255 1:255 110 0] [0x14
11 [2058 4] 100.000000] (REPLY_EXTRACLUSTER)
f 39.364561501 _20_ RTR  --- 0 AODV 76 [0 0 0 0] ------- [0:255 1:255 109 10] [0x4 12 [2058 4]
100.000000] (REPLY_INTRACLUSTER)
r 39.370180735 _9_ RTR  --- 0 AODV 76 [13a 9 14 800] ------- [0:255 1:255 109 10] [0x4 12
[2058 4] 100.000000] (REPLY_INTRACLUSTER)
f 39.374634735 _9_ RTR  --- 0 AODV 76 [13a 9 14 800] ------- [0:255 1:255 108 9] [0x4 13 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
```

```
r 39.379209968 _8_ RTR  --- 0 AODV 76 [13a 8 9 800] ------- [0:255 1:255 108 9] [0x4 13 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
f 39.383663968 _8_ RTR  --- 0 AODV 76 [13a 8 9 800] ------- [0:255 1:255 107 8] [0x4 14 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
r 39.389519201 _7_ RTR  --- 0 AODV 76 [13a 7 8 800] ------- [0:255 1:255 107 8] [0x4 14 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
f 39.393973201 _7_ RTR  --- 0 AODV 76 [13a 7 8 800] ------- [0:255 1:255 106 7] [0x4 15 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
r 39.398888435 _6_ RTR  --- 0 AODV 76 [13a 6 7 800] ------- [0:255 1:255 106 7] [0x4 15 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
f 39.403342435 _6_ RTR  --- 0 AODV 76 [13a 6 7 800] ------- [0:255 1:255 105 6] [0x4 16 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
r 39.408377668 _5_ RTR  --- 0 AODV 76 [13a 5 6 800] ------- [0:255 1:255 105 6] [0x4 16 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
f 39.412831668 _5_ RTR  --- 0 AODV 76 [13a 5 6 800] ------- [0:255 1:255 104 5] [0x4 17 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
r 39.418446901 _4_ RTR  --- 0 AODV 76 [13a 4 5 800] ------- [0:255 1:255 104 5] [0x4 17 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
f 39.422900901 _4_ RTR  --- 0 AODV 76 [13a 4 5 800] ------- [0:255 1:255 103 4] [0x4 18 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
r 39.428436135 _3_ RTR  --- 0 AODV 76 [13a 3 4 800] ------- [0:255 1:255 103 4] [0x4 18 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
f 39.432890135 _3_ RTR  --- 0 AODV 76 [13a 3 4 800] ------- [0:255 1:255 102 3] [0x4 19 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
r 39.438265368 _2_ RTR  --- 0 AODV 76 [13a 2 3 800] ------- [0:255 1:255 102 3] [0x4 19 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
f 39.442719368 _2_ RTR  --- 0 AODV 76 [13a 2 3 800] ------- [0:255 1:255 101 2] [0x4 20 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
r 39.447894601 _1_ RTR  --- 0 AODV 76 [13a 1 2 800] ------- [0:255 1:255 101 2] [0x4 20 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
f 39.452348601 _1_ RTR  --- 0 AODV 76 [13a 1 2 800] ------- [0:255 1:255 100 1] [0x4 21 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
r 39.457503835 _0_ RTR  --- 0 AODV 76 [13a 0 1 800] ------- [0:255 1:255 100 1] [0x4 21 [2058
4] 100.000000] (REPLY_INTRACLUSTER)
f 39.462370835 _0_ RTR  --- 0 cbr 532 [0 0 0 0] ------- [1:0 2058:0 120 2] [0] 0 0
r 39.467938935 _1_ RTR  --- 0 cbr 532 [13a 1 0 800] ------- [1:0 2058:0 120 2] [0] 1 0
s 39.467938935 _1_ RTR  --- 0 cbr 532 [13a 1 0 800] ------- [1:0 2058:0 119 3] [0] 1 0
r 39.474111035 _2_ RTR  --- 0 cbr 532 [13a 2 1 800] ------- [1:0 2058:0 119 3] [0] 2 0
f 39.474111035 _2_ RTR  --- 0 cbr 532 [13a 2 1 800] ------- [1:0 2058:0 118 4] [0] 2 0
r 39.479823135 _3_ RTR  --- 0 cbr 532 [13a 3 2 800] ------- [1:0 2058:0 118 4] [0] 3 0
f 39.479823135 _3_ RTR  --- 0 cbr 532 [13a 3 2 800] ------- [1:0 2058:0 117 5] [0] 3 0
r 39.486095235 _4_ RTR  --- 0 cbr 532 [13a 4 3 800] ------- [1:0 2058:0 117 5] [0] 4 0
f 39.486095235 _4_ RTR  --- 0 cbr 532 [13a 4 3 800] ------- [1:0 2058:0 116 6] [0] 4 0
r 39.492147335 _5_ RTR  --- 0 cbr 532 [13a 5 4 800] ------- [1:0 2058:0 116 6] [0] 5 0
f 39.492147335 _5_ RTR  --- 0 cbr 532 [13a 5 4 800] ------- [1:0 2058:0 115 7] [0] 5 0
r 39.498219435 _6_ RTR  --- 0 cbr 532 [13a 6 5 800] ------- [1:0 2058:0 115 7] [0] 6 0
f 39.498219435 _6_ RTR  --- 0 cbr 532 [13a 6 5 800] ------- [1:0 2058:0 114 8] [0] 6 0
r 39.504151535 _7_ RTR  --- 0 cbr 532 [13a 7 6 800] ------- [1:0 2058:0 114 8] [0] 7 0
f 39.504151535 _7_ RTR  --- 0 cbr 532 [13a 7 6 800] ------- [1:0 2058:0 113 9] [0] 7 0
r 39.509903635 _8_ RTR  --- 0 cbr 532 [13a 8 7 800] ------- [1:0 2058:0 113 9] [0] 8 0
f 39.509903635 _8_ RTR  --- 0 cbr 532 [13a 8 7 800] ------- [1:0 2058:0 112 10] [0] 8 0
r 39.516075735 _9_ RTR  --- 0 cbr 532 [13a 9 8 800] ------- [1:0 2058:0 112 10] [0] 9 0
f 39.516075735 _9_ RTR  --- 0 cbr 532 [13a 9 8 800] ------- [1:0 2058:0 111 0] [0] 9 0
r 39.522307835 _20_ RTR  --- 0 cbr 532 [13a 14 9 800] ------- [1:0 2058:0 111 0] [0] 10 0
f 81.248569501 _20_ RTR  --- 0 cbr 532 [13a 14 9 800] ------- [1:0 2058:0 110 2048] [0] 10 0
r 81.254059501 _21_ RTR  --- 0 cbr 532 [13a 15 14 800] ------- [1:0 2058:0 110 2048] [0] 11 0
f 81.260635501 _21_ RTR  --- 0 cbr 532 [13a 15 14 800] ------- [1:0 2058:0 109 2049] [0] 11 0
r 81.266443601 _10_ RTR  --- 0 cbr 532 [13a a 15 800] ------- [1:0 2058:0 109 2049] [0] 12 0
f 81.266443601 _10_ RTR  --- 0 cbr 532 [13a a 15 800] ------- [1:0 2058:0 108 2050] [0] 12 0
r 81.272515701 _11_ RTR  --- 0 cbr 532 [13a b a 800] ------- [1:0 2058:0 108 2050] [0] 13 0
f 81.272515701 _11_ RTR  --- 0 cbr 532 [13a b a 800] ------- [1:0 2058:0 107 2051] [0] 13 0
r 81.278567801 _12_ RTR  --- 0 cbr 532 [13a c b 800] ------- [1:0 2058:0 107 2051] [0] 14 0
f 81.278567801 _12_ RTR  --- 0 cbr 532 [13a c b 800] ------- [1:0 2058:0 106 2052] [0] 14 0
r 81.284379901 _13_ RTR  --- 0 cbr 532 [13a d c 800] ------- [1:0 2058:0 106 2052] [0] 15 0
f 81.284379901 _13_ RTR  --- 0 cbr 532 [13a d c 800] ------- [1:0 2058:0 105 2053] [0] 15 0
r 81.290572001 _14_ RTR  --- 0 cbr 532 [13a e d 800] ------- [1:0 2058:0 105 2053] [0] 16 0
f 81.290572001 _14_ RTR  --- 0 cbr 532 [13a e d 800] ------- [1:0 2058:0 104 2054] [0] 16 0
r 81.296544101 _15_ RTR  --- 0 cbr 532 [13a f e 800] ------- [1:0 2058:0 104 2054] [0] 17 0
f 81.296544101 _15_ RTR  --- 0 cbr 532 [13a f e 800] ------- [1:0 2058:0 103 2055] [0] 17 0
r 81.302676201 _16_ RTR  --- 0 cbr 532 [13a 10 f 800] ------- [1:0 2058:0 103 2055] [0] 18 0
f 81.302676201 _16_ RTR  --- 0 cbr 532 [13a 10 f 800] ------- [1:0 2058:0 102 2056] [0] 18 0
r 81.308488301 _17_ RTR  --- 0 cbr 532 [13a 11 10 800] ------- [1:0 2058:0 102 2056] [0] 19 0
f 81.308488301 _17_ RTR  --- 0 cbr 532 [13a 11 10 800] ------- [1:0 2058:0 101 2057] [0] 19 0
r 81.314300401 _18_ RTR  --- 0 cbr 532 [13a 12 11 800] ------- [1:0 2058:0 101 2057] [0] 20 0
f 81.314300401 _18_ RTR  --- 0 cbr 532 [13a 12 11 800] ------- [1:0 2058:0 100 2058] [0] 20 0
r 81.320532501 _19_ AGT  --- 0 cbr 532 [13a 13 12 800] ------- [1:0 2058:0 100 2058] [0] 21 0
s 100.000000000 _0_ AGT  --- 1 cbr 512 [0 0 0 0] ------- [1:0 2058:0 32 0] [1] 0 0
```

```
r 100.000000000 _0_ RTR  --- 1 cbr 512 [0 0 0 0] ------- [1:0 2058:0 32 0] [1] 0 0
f 100.000000000 _0_ RTR  --- 1 cbr 532 [0 0 0 0] ------- [1:0 2058:0 120 2] [1] 0 0
r 100.006048100 _1_ RTR  --- 1 cbr 532 [13a 1 0 800] ------- [1:0 2058:0 120 2] [1] 1 0
s 100.006048100 _1_ RTR  --- 1 cbr 532 [13a 1 0 800] ------- [1:0 2058:0 119 3] [1] 1 0
r 100.012340200 _2_ RTR  --- 1 cbr 532 [13a 2 1 800] ------- [1:0 2058:0 119 3] [1] 2 0
f 100.012340200 _2_ RTR  --- 1 cbr 532 [13a 2 1 800] ------- [1:0 2058:0 118 4] [1] 2 0
r 100.018072300 _3_ RTR  --- 1 cbr 532 [13a 3 2 800] ------- [1:0 2058:0 118 4] [1] 3 0
f 100.018072300 _3_ RTR  --- 1 cbr 532 [13a 3 2 800] ------- [1:0 2058:0 117 5] [1] 3 0
r 100.024004400 _4_ RTR  --- 1 cbr 532 [13a 4 3 800] ------- [1:0 2058:0 117 5] [1] 4 0
f 100.024004400 _4_ RTR  --- 1 cbr 532 [13a 4 3 800] ------- [1:0 2058:0 116 6] [1] 4 0
r 100.030296500 _5_ RTR  --- 1 cbr 532 [13a 5 4 800] ------- [1:0 2058:0 116 6] [1] 5 0
f 100.030296500 _5_ RTR  --- 1 cbr 532 [13a 5 4 800] ------- [1:0 2058:0 115 7] [1] 5 0
r 100.036148600 _6_ RTR  --- 1 cbr 532 [13a 6 5 800] ------- [1:0 2058:0 115 7] [1] 6 0
f 100.036148600 _6_ RTR  --- 1 cbr 532 [13a 6 5 800] ------- [1:0 2058:0 114 8] [1] 6 0
r 100.042340700 _7_ RTR  --- 1 cbr 532 [13a 7 6 800] ------- [1:0 2058:0 114 8] [1] 7 0
f 100.042340700 _7_ RTR  --- 1 cbr 532 [13a 7 6 800] ------- [1:0 2058:0 113 9] [1] 7 0
r 100.048172800 _8_ RTR  --- 1 cbr 532 [13a 8 7 800] ------- [1:0 2058:0 113 9] [1] 8 0
f 100.048172800 _8_ RTR  --- 1 cbr 532 [13a 8 7 800] ------- [1:0 2058:0 112 10] [1] 8 0
r 100.054364900 _9_ RTR  --- 1 cbr 532 [13a 9 8 800] ------- [1:0 2058:0 112 10] [1] 9 0
f 100.054364900 _9_ RTR  --- 1 cbr 532 [13a 9 8 800] ------- [1:0 2058:0 111 0] [1] 9 0
r 100.060277000 _20_ RTR  --- 1 cbr 532 [13a 14 9 800] ------- [1:0 2058:0 111 0] [1] 10 0
f 100.066853000 _20_ RTR  --- 1 cbr 532 [13a 14 9 800] ------- [1:0 2058:0 110 2048] [1] 10 0
r 100.072723000 _21_ RTR  --- 1 cbr 532 [13a 15 14 800] ------- [1:0 2058:0 110 2048] [1] 11 0
f 100.079299000 _21_ RTR  --- 1 cbr 532 [13a 15 14 800] ------- [1:0 2058:0 109 2049] [1] 11 0
r 100.085187100 _10_ RTR  --- 1 cbr 532 [13a a 15 800] ------- [1:0 2058:0 109 2049] [1] 12 0
f 100.085187100 _10_ RTR  --- 1 cbr 532 [13a a 15 800] ------- [1:0 2058:0 108 2050] [1] 12 0
r 100.091159200 _11_ RTR  --- 1 cbr 532 [13a b a 800] ------- [1:0 2058:0 108 2050] [1] 13 0
f 100.091159200 _11_ RTR  --- 1 cbr 532 [13a b a 800] ------- [1:0 2058:0 107 2051] [1] 13 0
r 100.097251300 _12_ RTR  --- 1 cbr 532 [13a c b 800] ------- [1:0 2058:0 107 2051] [1] 14 0
f 100.097251300 _12_ RTR  --- 1 cbr 532 [13a c b 800] ------- [1:0 2058:0 106 2052] [1] 14 0
r 100.103483400 _13_ RTR  --- 1 cbr 532 [13a d c 800] ------- [1:0 2058:0 106 2052] [1] 15 0
f 100.103483400 _13_ RTR  --- 1 cbr 532 [13a d c 800] ------- [1:0 2058:0 105 2053] [1] 15 0
r 100.109635500 _14_ RTR  --- 1 cbr 532 [13a e d 800] ------- [1:0 2058:0 105 2053] [1] 16 0
f 100.109635500 _14_ RTR  --- 1 cbr 532 [13a e d 800] ------- [1:0 2058:0 104 2054] [1] 16 0
r 100.115347600 _15_ RTR  --- 1 cbr 532 [13a f e 800] ------- [1:0 2058:0 104 2054] [1] 17 0
f 100.115347600 _15_ RTR  --- 1 cbr 532 [13a f e 800] ------- [1:0 2058:0 103 2055] [1] 17 0
r 100.121439700 _16_ RTR  --- 1 cbr 532 [13a 10 f 800] ------- [1:0 2058:0 103 2055] [1] 18 0
f 100.121439700 _16_ RTR  --- 1 cbr 532 [13a 10 f 800] ------- [1:0 2058:0 102 2056] [1] 18 0
r 100.127631800 _17_ RTR  --- 1 cbr 532 [13a 11 10 800] ------- [1:0 2058:0 102 2056] [1] 19 0
f 100.127631800 _17_ RTR  --- 1 cbr 532 [13a 11 10 800] ------- [1:0 2058:0 101 2057] [1] 19 0
r 100.133463900 _18_ RTR  --- 1 cbr 532 [13a 12 11 800] ------- [1:0 2058:0 101 2057] [1] 20 0
f 100.133463900 _18_ RTR  --- 1 cbr 532 [13a 12 11 800] ------- [1:0 2058:0 100 2058] [1] 20 0
r 100.139316000 _19_ AGT  --- 1 cbr 532 [13a 13 12 800] ------- [1:0 2058:0 100 2058] [1] 21 0
```

ANEXO II

Parte do conteúdo do arquivo **aodv_packet.h**:

```
/*
Copyright (c) 1997, 1998 Carnegie Mellon University.  All Rights
Reserved.

Permission to use, copy, modify, and distribute this
software and its documentation is hereby granted (including for
commercial or for-profit use), provided that both the copyright notice and this permission
notice appear in all copies of the software, derivative works, or modified versions, and any
portions thereof, and that both notices appear in supporting documentation, and that credit is
given to Carnegie Mellon University in all publications reporting on direct or indirect use of
this code or its derivatives.

ALL CODE, SOFTWARE, PROTOCOLS, AND ARCHITECTURES DEVELOPED BY THE CMU
MONARCH PROJECT ARE EXPERIMENTAL AND ARE KNOWN TO HAVE BUGS, SOME OF
WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
SOFTWARE OR OTHER INTELLECTUAL PROPERTY IN ITS ``AS IS'' CONDITION,
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR
INTELLECTUAL PROPERTY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

Carnegie Mellon encourages (but does not require) users of this
software or intellectual property to return any improvements or
extensions that they make, and to grant Carnegie Mellon the rights to redistribute these
changes without encumbrance.

The AODV code developed by the CMU/MONARCH group was optimized and tuned by Samir Das and
Mahesh Marina, University of Cincinnati. The work was partially done in Sun Microsystems.
*/
#include <openssl/ec.h>
#include <openssl/ecdsa.h>
/* ======================================================================
   Packet Formats...
   ====================================================================== */
#define AODVTYPE_HELLO        0x01
#define AODVTYPE_RREQ_IC      0x02
#define AODVTYPE_RREP_IC      0x04
#define AODVTYPE_RRINI        0x06
#define AODVTYPE_RERR         0x08
#define AODVTYPE_RREP_ACK     0x10
#define AODVTYPE_RREQ_EC      0x12
#define AODVTYPE_RREP_EC      0x14

/*
 * AODV Routing Protocol Header Macros
 */
#define HDR_AODV(p)           ((struct hdr_aodv*)hdr_aodv::access(p))
#define HDR_AODV_REQUEST_IC(p)  ((struct hdr_aodv_request_ic*)hdr_aodv::access(p))
#define HDR_AODV_REPLY_IC(p)  ((struct hdr_aodv_reply_ic*)hdr_aodv::access(p))
#define HDR_AODV_REQUEST_EC(p)  ((struct hdr_aodv_request_ec*)hdr_aodv::access(p))
#define HDR_AODV_REPLY_EC(p)  ((struct hdr_aodv_reply_ec*)hdr_aodv::access(p))
#define HDR_AODV_RRINI(p)     ((struct hdr_aodv_rrini*)hdr_aodv::access(p))
#define HDR_AODV_ERROR(p)     ((struct hdr_aodv_error*)hdr_aodv::access(p))
#define HDR_AODV_RREP_ACK(p)  ((struct hdr_aodv_rrep_ack*)hdr_aodv::access(p))
struct hdr_aodv {
        u_int8_t        ah_type;
       static int offset_; // required by PacketHeaderManager
       inline static int& offset() { return offset_; }
       inline static hdr_aodv* access(const Packet* p) {
              return (hdr_aodv*) p->access(offset_);
       }
};
```

```
struct hdr_aodv_request_ic {
        u_int8_t        rq_type;      // Packet Type
        u_int8_t        reserved[2];
        u_int8_t        rq_hop_count;   // Hop Count
        u_int32_t       rq_bcast_id;    // Broadcast ID

        nsaddr_t        rq_dst;         // Destination IP Address
        u_int32_t       rq_dst_seqno;   // Destination Sequence Number
        nsaddr_t        rq_src;         // Source IP Address
        u_int32_t       rq_src_seqno;   // Source Sequence Number

        double          rq_timestamp;   // when REQUEST sent;
                                        // used to compute route discovery latency
        char   rq_signature[32];
  inline int size() {
  int sz = 0;
       sz = 7*sizeof(u_int32_t) + 32;  // 32 = 2x16 bytes blocks for SHA1 (20 bytes) RC5
encryption
       assert (sz >= 0);
       return sz;
  }
};
struct hdr_aodv_request_ec {
        u_int8_t        rq_type;      // Packet Type
        u_int8_t        reserved[2];
        u_int8_t        rq_hop_count;   // Hop Count
        u_int32_t       rq_bcast_id;    // Broadcast ID

        nsaddr_t        rq_dst;         // Destination IP Address
        u_int32_t       rq_dst_seqno;   // Destination Sequence Number
        nsaddr_t        rq_src;         // Source IP Address
        u_int32_t       rq_src_seqno;   // Source Sequence Number

        double          rq_timestamp;   // when REQUEST sent;
                                       // used to compute route discovery latency
        char            rq_top_hash[20]; // top hash to control the hop count field
        char            rq_hash[20];    // hash to be re-calculated on each hop
        ECDSA_SIG       *rq_signature;  //the message's signature
        EC_KEY          *rq_public_key; // as EC_KEY has a variable length, a pointer should
be used instead

  inline int size() {
  int sz = 0;
       sz = 7*sizeof(u_int32_t) + 108; // 108 = rq_top_hash, rq_hash, rq_public_key,
rq_signature
       assert (sz >= 0);
       return sz;
  }
};

struct hdr_aodv_reply_ic {
        u_int8_t        rp_type;        // Packet Type
        u_int8_t        reserved[2];
        u_int8_t        rp_hop_count;           // Hop Count
        nsaddr_t        rp_dst;                 // Destination IP Address
        u_int32_t       rp_dst_seqno;           // Destination Sequence Number
        nsaddr_t        rp_src;                 // Source IP Address
        double         rp_lifetime;            // Lifetime

        double          rp_timestamp;           // when corresponding REQ sent;
                                               // used to compute route discovery latency
        char   rp_signature[32];

  inline int size() {
  int sz = 0;
       sz = 6*sizeof(u_int32_t) + 32;  // 32 = 2x16 block RC5 encryption of SHA1(20bytes)
       assert (sz >= 0);
       return sz;
  }

};

struct hdr_aodv_reply_ec {
        u_int8_t        rp_type;        // Packet Type
        u_int8_t        reserved[2];
        u_int8_t        rp_hop_count;           // Hop Count
        nsaddr_t        rp_dst;                 // Destination IP Address
```

```
        u_int32_t       rp_dst_seqno;           // Destination Sequence Number
        nsaddr_t        rp_src;                 // Source IP Address
        double         rp_lifetime;            // Lifetime

        double          rp_timestamp;           // when corresponding REQ sent;
                                               // used to compute route discovery latency
        char   rp_top_hash[20]; // top hash to control the hop count field
        char   rp_hash[20];    // hash to be re-calculated on each hop
        ECDSA_SIG       *rp_signature;  //the message's signature
        EC_KEY          *rp_public_key; // 160 bits public key



  inline int size() {
  int sz = 0;
        sz = 6*sizeof(u_int32_t) + 108;
        assert (sz >= 0);
        return sz;
  }

};

struct hdr_aodv_error {
        u_int8_t        re_type;                // Type
        u_int8_t        reserved[2];            // Reserved
        u_int8_t        DestCount;                 // DestCount
        // List of Unreachable destination IP addresses and sequence numbers
        nsaddr_t        unreachable_dst[AODV_MAX_ERRORS];
        u_int32_t       unreachable_dst_seqno[AODV_MAX_ERRORS];

  inline int size() {
  int sz = 0;
        sz = (DestCount*2 + 1)*sizeof(u_int32_t);
        assert(sz);
         return sz;
  }

};

struct hdr_aodv_rrep_ack {
       u_int8_t       rpack_type;
       u_int8_t       reserved;
};

/* hdr_aodv_rrini  - Route Request Initiator  */
struct hdr_aodv_rrini {
   u_int8_t        rri_type;  // Packet Type
   u_int32_t       rri_bcast_id;    // Broadcast ID
   nsaddr_t        rri_dst;         // Destination IP Address
   nsaddr_t        rri_src;         // Source IP Address
   char            rri_signature[32];

  inline int size() {
  int sz = 0;
  sz = 48;//7*sizeof(u_int32_t);
  assert (sz >= 0);
  return sz;
  }
};

// for size calculation of header-space reservation
union hdr_all_aodv {
  hdr_aodv          ah;
  hdr_aodv_request_ic  rreq_ic;
  hdr_aodv_reply_ic    rrep_ic;
  hdr_aodv_request_ec  rreq_ec;
  hdr_aodv_reply_ec    rrep_ec;
  hdr_aodv_error    rerr;
  hdr_aodv_rrep_ack rrep_ack;
  hdr_aodv_rrini rrini;
};

#endif /* __aodv_packet_h__ */
```

Parte do conteúdo do arquivo **aodv.h**:

```
/*
Copyright (c) 1997, 1998 Carnegie Mellon University.  All Rights
Reserved.

Permission to use, copy, modify, and distribute this
software and its documentation is hereby granted (including for
commercial or for-profit use), provided that both the copyright notice and this permission
notice appear in all copies of the software, derivative works, or modified versions, and any
portions thereof, and that both notices appear in supporting documentation, and that credit is
given to Carnegie Mellon University in all publications reporting on direct or indirect use of
this code or its derivatives.

ALL CODE, SOFTWARE, PROTOCOLS, AND ARCHITECTURES DEVELOPED BY THE CMU
MONARCH PROJECT ARE EXPERIMENTAL AND ARE KNOWN TO HAVE BUGS, SOME OF
WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
SOFTWARE OR OTHER INTELLECTUAL PROPERTY IN ITS ``AS IS'' CONDITION,
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR
INTELLECTUAL PROPERTY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

Carnegie Mellon encourages (but does not require) users of this
software or intellectual property to return any improvements or
extensions that they make, and to grant Carnegie Mellon the rights to redistribute these
changes without encumbrance.

The AODV code developed by the CMU/MONARCH group was optimized and tuned by Samir Das and
Mahesh Marina, University of Cincinnati. The work was partially done in Sun Microsystems.

*/

#ifndef __aodv_h__
#define __aodv_h__

#include <openssl/ecdsa.h>
#include <openssl/evp.h>

#include <cmu-trace.h>
#include <priqueue.h>
#include <aodv/aodv_rtable.h>
#include <aodv/aodv_rqueue.h>
#include <classifier/classifier-port.h>
#define AODV_LOCAL_REPAIR
#define AODV_LINK_LAYER_DETECTION
#define AODV_USE_LL_METRIC

class AODV;

#define CH_POWER               1     // Cluster Head is CH_POWER times faster than a common
sensor
                                     // 1 =  8 MHz
                                     // 2 = 16 MHz
                                     // 4 = 32 MHz
                                     // 8 = 64 MHz
#define MY_ROUTE_TIMEOUT       100                         // 100 seconds shakal: was 10
#define ACTIVE_ROUTE_TIMEOUT   100                         // 50 seconds   was 10
#define REV_ROUTE_LIFE         100                         // 5  seconds was 6
#define BCAST_ID_SAVE          100                         // 3 seconds was 6
#define RREQ_RETRIES           10// : was 3
// timeout after doing network-wide search RREQ_RETRIES times
#define MAX_RREQ_TIMEOUT      100.0 //sec    shakal: was 10

/* Various constants used for the expanding ring search */
#define TTL_START     5
#define TTL_THRESHOLD 7
#define TTL_INCREMENT 2
```

```
// This should be somewhat related to arp timeout
#define NODE_TRAVERSAL_TIME     10 //shakal: was 0.03             // 30 ms
#define LOCAL_REPAIR_WAIT_TIME  50 //shakal: was 0.15 //sec

// Should be set by the user using best guess (conservative)
#define NETWORK_DIAMETER        120             // 30 hops: was 30 (shakal)

// Must be larger than the time difference between a node propagates a route
// request and gets the route reply back.

//#define RREP_WAIT_TIME     (3 * NODE_TRAVERSAL_TIME * NETWORK_DIAMETER) // ms
//#define RREP_WAIT_TIME     (2 * REV_ROUTE_LIFE)  // seconds
#define RREP_WAIT_TIME         100.0  // sec   shakal: was 1.0

#define ID_NOT_FOUND    0x00
#define ID_FOUND        0x01
//#define INFINITY        0xff

// The followings are used for the forward() function. Controls pacing.
#define DELAY 1.0           // random delay
#define NO_DELAY -1.0       // no delay

// think it should be 30 ms
#define ARP_DELAY 0.01      // fixed delay to keep arp happy


#define HELLO_INTERVAL          100               // 1000 ms shakal: was 1
#define ALLOWED_HELLO_LOSS      3               // packets
#define BAD_LINK_LIFETIME       300               // 3000 ms shakal: was 3
#define MaxHelloInterval        (1.25 * HELLO_INTERVAL)
#define MinHelloInterval        (0.75 * HELLO_INTERVAL)

/*
  Timers (Broadcast ID, Hello, Neighbor Cache, Route Cache)
*/
class BroadcastTimer : public Handler {
public:
        BroadcastTimer(AODV* a) : agent(a) {}
        void   handle(Event*);
private:
        AODV    *agent;
       Event   intr;
};

class HelloTimer : public Handler {
public:
        HelloTimer(AODV* a) : agent(a) {}
        void   handle(Event*);
private:
        AODV    *agent;
       Event   intr;
};

class NeighborTimer : public Handler {
public:
        NeighborTimer(AODV* a) : agent(a) {}
        void   handle(Event*);
private:
        AODV    *agent;
       Event   intr;
};

class RouteCacheTimer : public Handler {
public:
        RouteCacheTimer(AODV* a) : agent(a) {}
        void   handle(Event*);
private:
        AODV    *agent;
       Event   intr;
};

class LocalRepairTimer : public Handler {
public:
        LocalRepairTimer(AODV* a) : agent(a) {}
        void   handle(Event*);
private:
        AODV    *agent;
```

```
        Event   intr;
};
// shakal. Used to store RREQ_EC received
class Request_EC_Container{
 public:
        Request_EC_Container();
        void     addRequest_ec(nsaddr_t dst, Packet *p);
        Packet*  getRequest_ec(nsaddr_t dst);
        bool     existRequest_ec(nsaddr_t dst);
        void clear_ec();
 private:
        Packet   *pacote;
        nsaddr_t destino;
};

/*
  Broadcast ID Cache
*/
class BroadcastID {
        friend class AODV;
 public:
        BroadcastID(nsaddr_t i, u_int32_t b) { src = i; id = b;  }
 protected:
        LIST_ENTRY(BroadcastID) link;
        nsaddr_t        src;
        u_int32_t       id;
        double          expire;         // now + BCAST_ID_SAVE s
};

LIST_HEAD(aodv_bcache, BroadcastID);


/*
  The Routing Agent
*/
class AODV: public Agent {

  /*
   * make some friends first
   */

        friend class aodv_rt_entry;
        friend class BroadcastTimer;
        friend class HelloTimer;
        friend class NeighborTimer;
        friend class RouteCacheTimer;
        friend class LocalRepairTimer;
        friend class Request_EC_Container;


 public:
        AODV(nsaddr_t id);

        void             recv(Packet *p, Handler *);

 protected:
        int              command(int, const char *const *);
        int              initialized() { return 1 && target_; }

        /*
         * Route Table Management
         */
        void             rt_resolve(Packet *p);
        void             rt_update(aodv_rt_entry *rt, u_int32_t seqnum,
        u_int16_t        metric,
        nsaddr_t         nexthop,
        double           expire_time);
        void             rt_down(aodv_rt_entry *rt);
        void             local_rt_repair(aodv_rt_entry *rt, Packet *p);
 public:
        void             rt_ll_failed(Packet *p);
        void             handle_link_failure(nsaddr_t id);
 protected:
        void             rt_purge(void);

        void             enque(aodv_rt_entry *rt, Packet *p);
        Packet*          deque(aodv_rt_entry *rt);
```

```
        /*
         * Neighbor Management
         */
        void             nb_insert(nsaddr_t id);
        AODV_Neighbor*   nb_lookup(nsaddr_t id);
        void             nb_delete(nsaddr_t id);
        void             nb_purge(void);

        /*
         * Broadcast ID Management
         */

        void             id_insert(nsaddr_t id, u_int32_t bid);
        bool             id_lookup(nsaddr_t id, u_int32_t bid);
        void             id_purge(void);

        /*
         * Packet TX Routines
         */
        void             forward(aodv_rt_entry *rt, Packet *p, double delay);
        void             sendHello(void);
        void             sendRequest_ic(nsaddr_t dst, double delay_sec);
        void             sendRequest_ec(nsaddr_t dst, Packet *pkt, int ttl, double delay_sec);
        void             sendRRIni(nsaddr_t dst, double delay_sec);  /*shakal*/
        void             sendReply_ic(nsaddr_t ipdst, u_int32_t hop_count,
                                      nsaddr_t rpdst, u_int32_t rpseq,
                                      u_int32_t lifetime, double timestamp, int ttl, double
delay_sec);
        void             sendReply_ec(nsaddr_t ipdst, u_int32_t hop_count,
                                      nsaddr_t rpdst, u_int32_t rpseq,
                                      u_int32_t lifetime, double timestamp, int ttl, double
delay_sec);
        void             sendError(Packet *p, bool jitter = true);

        /*
         * Packet RX Routines
         */
        void             recvAODV(Packet *p);
        void             recvHello(Packet *p);
        /*shakal was here*/
        void             recvRequest_ic(Packet *p);// IC stands for Intra Cluster
        void             recvReply_ic(Packet *p); // ie, sensor to sensor, sensor to CH
        void             recvRequest_ec(Packet *p);// EC stands for Extra Cluster
        void             recvReply_ec(Packet *p);
        void             recvRRIni(Packet *p); /*shakal*/
        void             recvError(Packet *p);

       /*
        * History management
        */

       double           PerHopTime(aodv_rt_entry *rt);


        nsaddr_t         index;                  // IP Address of this node
        u_int32_t        seqno;                  // Sequence Number
        int              bid;                    // Broadcast ID

        aodv_rtable      rthead;                 // routing table
        aodv_ncache      nbhead;                 // Neighbor Cache
        aodv_bcache      bihead;                 // Broadcast ID Cache

        /*
         * Timers
         */
        BroadcastTimer   btimer;
        HelloTimer       htimer;
        NeighborTimer    ntimer;
        RouteCacheTimer  rtimer;
        LocalRepairTimer lrtimer;

        /*
         * Routing Table
         */
        aodv_rtable      rtable;
        /*
```

```
         *  A "drop-front" queue used by the routing layer to buffer
         *  packets to which it does not have a route.
         */
        aodv_rqueue      rqueue;

        /*
         * A mechanism for logging the contents of the routing
         * table.
         */
        Trace            *logtarget;

        /*
         * A pointer to the network interface queue that sits
         * between the "classifier" and the "link layer".
         */
        PriQueue         *ifqueue;

        /*
         * Logging stuff
         */
        void             log_link_del(nsaddr_t dst);
        void             log_link_broke(Packet *p);
        void             log_link_kept(nsaddr_t dst);

       /* for passing packets up to agents */
       PortClassifier   *dmux_;

       // Cluster communication Management
       Request_EC_Container last_rreq_ec;

       /*
       *  Security Management
       */
        EC_KEY           *ec_key;
        char             group_key[40];
        char             dh_key[40];
        double           hold_until;

        //IC - Functions
        void             calculate_HASH_ic(Packet *p,double *delay,char digest[20]);
        void             do_sign_ic(Packet *p, char digest[20], double *delay);
        bool             is_signature_valid_ic(Packet *p, double *delay);

        //EC - Functions
        void             calculate_HASH_ec(Packet *p,double *delay, char digest[20]);
        void             calculate_HASH_hop_count_ec(Packet *p,double *delay, char
digest[20]);
        void             calculate_HASH_over_HASH_ec(char in[20],double *delay,char out[20]);
        void             calculate_TOP_HASH_ec(char in[20], int times, double *delay,char
out[20]);
        void             do_sign_ec(Packet *p, char digest[20], double *delay);
        bool             is_signature_valid_ec(Packet *p,double *delay);
        void             calculate_DH_KEY_ec(EC_KEY *public_key,double *delay);

        //RRINI
        void             calculate_HASH_rrini(Packet *p,double *delay, char digest[20]);
        void             do_sign_rrini(Packet *p,double *delay);
        bool             is_signature_valid_rrini(Packet *p, double *delay);

};

#endif /* __aodv_h__ */
```

Parte do conteúdo do arquivo **aodv.cc**:

```
/*
Copyright (c) 1997, 1998 Carnegie Mellon University.  All Rights
Reserved.

Permission to use, copy, modify, and distribute this
software and its documentation is hereby granted (including for
commercial or for-profit use), provided that both the copyright notice and this permission
notice appear in all copies of the software, derivative works, or modified versions, and any
portions thereof, and that both notices appear in supporting documentation, and that credit is
given to Carnegie Mellon University in all publications reporting on direct or indirect use of
this code or its derivatives.

ALL CODE, SOFTWARE, PROTOCOLS, AND ARCHITECTURES DEVELOPED BY THE CMU
MONARCH PROJECT ARE EXPERIMENTAL AND ARE KNOWN TO HAVE BUGS, SOME OF
WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
SOFTWARE OR OTHER INTELLECTUAL PROPERTY IN ITS ``AS IS'' CONDITION,
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR
INTELLECTUAL PROPERTY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

Carnegie Mellon encourages (but does not require) users of this
software or intellectual property to return any improvements or
extensions that they make, and to grant Carnegie Mellon the rights to redistribute these
changes without encumbrance.

The AODV code developed by the CMU/MONARCH group was optimized and tuned by Samir Das and
Mahesh Marina, University of Cincinnati. The work was partially done in Sun Microsystems.
Modified for gratuitous replies by Anant Utgikar, 09/16/02.

*/
#include <math.h>

//#include <ip.h>

#include <aodv/aodv.h>
#include <aodv/aodv_packet.h>
#include <random.h>
#include <cmu-trace.h>


#include "../e_os.h"
#include <openssl/opensslconf.h>
#include <openssl/ecdsa.h>
#include <openssl/ec.h>
#include <openssl/ecdh.h>
#include <openssl/rc5.h>
#include <openssl/sha.h>



static const int KDF1_SHA1_len = 20;
static void *KDF1_SHA1(const void *in, size_t inlen, void *out, size_t *outlen){
  *outlen = SHA_DIGEST_LENGTH;
  return SHA1((const unsigned char *)in, inlen, (unsigned char *)out);
}

AODV::AODV(nsaddr_t id) : Agent(PT_AODV),
                          btimer(this), htimer(this), ntimer(this),
                          rtimer(this), lrtimer(this), rqueue(), last_rreq_ec() {   // shakal:
last_rreq_ec


  index = id;
  seqno = 2;
  bid = 1;
```

```
  LIST_INIT(&nbhead);
  LIST_INIT(&bihead);

  logtarget = 0;
  ifqueue = 0;
  //time necessary to perform cryptographic functions
  hold_until = 0.0;

  if((index%2048)==0){  //for interconnection, just CH's must have its key
    ec_key = EC_KEY_new_by_curve_name(NID_X9_62_prime192v1);
    if (!EC_KEY_generate_key(ec_key)){
      printf("\nErro ao gerar a chave");
      printf("\n index = %u \n",index);
    }
  }
  // this must be implemented through TCL file... for while, it's working ...
  if(floor(index/2048)==0){
    strcpy(group_key,"B51C8F5AFDEC7ACA11EAD5EA184732839C7B7502");
  }else if (floor(index/2048)==1){
    strcpy(group_key,"8975D4B711206697F02F16CA6385DA47A54BD373");
  }


}

void
AODV::rt_resolve(Packet *p) {
Packet *pkt = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
aodv_rt_entry *rt;
double delay_sec = hold_until - CURRENT_TIME;
if(delay_sec < 0.0) delay_sec = 0.0;
double pkt_size = ch->size() - IP_HDR_LEN; //size of data packet (CBR)
double aux;
double ch_power = 1.0;
bool sou_CH = (index%2048)==0;
   // checking if the request is addressed to another cluster
bool mesma_origem = (floor(index/2048))==(floor(ih->saddr()/2048));
bool mesmo_destino = (floor(index/2048))==(floor(ih->daddr()/2048));

if(sou_CH) ch_power = CH_POWER;

 /*
  *  Set the transmit failure callback.  That
  *  won't change.
  */
 ch->xmit_failure_ = aodv_rt_failed_callback;
 ch->xmit_failure_data_ = (void*) this;
       rt = rtable.rt_lookup(ih->daddr());
 if(rt == 0) {
         rt = rtable.rt_add(ih->daddr());
 }

 /*
  * If the route is up, forward the packet
  */

 if(rt->rt_flags == RTF_UP) {
   assert(rt->rt_hops != INFINITY2);
   if(sou_CH &&(mesma_origem || mesmo_destino)){
     aux = pkt_size / 64;
     aux = (double)ceil(aux);
     aux = aux * 0.000409; //RC5 decryption
     aux = aux / ch_power;
     delay_sec += aux;
     aux = pkt_size / 64;
     aux = (double)ceil(aux);
     aux = aux * 0.000413; //RC5 encryption
     aux = aux / ch_power;
     delay_sec += aux;
   }
   forward(rt, p, delay_sec);
 }
 /*
  *  if I am the source of the packet, then do a Route Request.
  */
```

```
        else if(ih->saddr() == index) {
    rqueue.enque(p);
    // shakal
    // checking if I`m the cluster head
    bool sou_CH = (index%2048)==0;
    // checking if the request is addressed to another cluster
    bool eh_pra_fora = (floor(index/2048))!=(floor(rt->rt_dst/2048));
    if (sou_CH){
      if (eh_pra_fora){
        sendRequest_ec(rt->rt_dst,pkt,NETWORK_DIAMETER,0.);
      }
      else{
        sendRRIni(rt->rt_dst,0.);
      }
    }
    else{
      sendRequest_ic(rt->rt_dst,0.);
    }
  }
  /*
   *    A local repair is in progress. Buffer the packet.
   */
  else if (rt->rt_flags == RTF_IN_REPAIR) {
    rqueue.enque(p);
  }

  /*
   * I am trying to forward a packet for someone else to which
   * I don't have a route.
   */
  else {
  Packet *rerr = Packet::alloc();
  struct hdr_aodv_error *re = HDR_AODV_ERROR(rerr);
  /*
   * For now, drop the packet and send error upstream.
   * Now the route errors are broadcast to upstream
   * neighbors - Mahesh 09/11/99
   */

    assert (rt->rt_flags == RTF_DOWN);
    re->DestCount = 0;
    re->unreachable_dst[re->DestCount] = rt->rt_dst;
    re->unreachable_dst_seqno[re->DestCount] = rt->rt_seqno;
    re->DestCount += 1;
#ifdef DEBUG
    fprintf(stderr, "%s: sending RERR...\n", __FUNCTION__);
#endif
    sendError(rerr, false);

    drop(p, DROP_RTR_NO_ROUTE);
  }

}

/*
  Packet Reception Routines
*/

void
AODV::recv(Packet *p, Handler*) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);

  assert(initialized());
  //assert(p->incoming == 0);
  // XXXXX NOTE: use of incoming flag has been depracated; In order to track direction of pkt
flow, direction_ in hdr_cmn is used instead. see packet.h for details.

  if(ch->ptype() == PT_AODV) {
    ih->ttl_ -= 1;
    recvAODV(p);
    return;
  }


  /*
   *  Must be a packet I'm originating...
```

```
  */
if((ih->saddr() == index) && (ch->num_forwards() == 0)) {
 /*
  * Add the IP Header
  */
   ch->size() += IP_HDR_LEN;
   // Added by Parag Dadhania && John Novatnack to handle broadcasting
   if ( (u_int32_t)ih->daddr() != IP_BROADCAST)
     ih->ttl_ = NETWORK_DIAMETER;
}
 /*
  *  I received a packet that I sent.  Probably
  *  a routing loop.
  */
else if(ih->saddr() == index) {
   drop(p, DROP_RTR_ROUTE_LOOP);
   return;
 }
 /*
  *  Packet I'm forwarding...
  */
 else {
 /*
  *  Check the TTL.  If it is zero, then discard.
  */
   if(--ih->ttl_ == 0) {
     drop(p, DROP_RTR_TTL);
     return;
   }
 }
// Added by Parag Dadhania && John Novatnack to handle broadcasting
 if ( (u_int32_t)ih->daddr() != IP_BROADCAST)
   rt_resolve(p);
 else
   forward((aodv_rt_entry*) 0, p, NO_DELAY);
}


void
AODV::recvAODV(Packet *p) {
struct hdr_aodv *ah = HDR_AODV(p);
struct hdr_ip *ih = HDR_IP(p);

 assert(ih->sport() == RT_PORT);
 assert(ih->dport() == RT_PORT);
 if (ih->sport() !=RT_PORT) printf("this is just to avoid: aodv/aodv.cc:687: warning: unused
variable ih");

 /*
  * Incoming Packets.
  */
 switch(ah->ah_type) {

 case AODVTYPE_RREQ_IC:
   recvRequest_ic(p);
   break;

 case AODVTYPE_RREQ_EC:
   recvRequest_ec(p);
   break;

 case AODVTYPE_RREP_IC:
   recvReply_ic(p);
   break;

    case AODVTYPE_RREP_EC:
   recvReply_ec(p);
   break;

 case AODVTYPE_RERR:
   recvError(p);
   break;

 case AODVTYPE_HELLO:
   recvHello(p);
   break;
```

```
 case AODVTYPE_RRINI:
   recvRRIni(p);
   break;

 default:
   fprintf(stderr, "Invalid AODV type (%x)\n", ah->ah_type);
   exit(1);
 }

}
void
AODV::recvRequest_ic(Packet *p) {
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_request_ic *rq = HDR_AODV_REQUEST_IC(p);
struct hdr_aodv_request_ec *rq_ec;
// this is necessary to verify if some cryptographic function is still being performed...
double delay_sec = hold_until - CURRENT_TIME;
if(delay_sec < 0.0) delay_sec = 0.0;
double aux = 0.0;
aodv_rt_entry *rt;

/*
 *  Secure Management
 */

if(!is_signature_valid_ic(p,&delay_sec)){  // if the signature is wrong, discard the packet
  Packet::free(p);                        // else, delay_sec contains the necessary time
  return;                               // to perform the cryptographic functions
}



if (rt && (rt->rt_hops != INFINITY2) && (rt->rt_seqno >= rq->rq_dst_seqno) ) {

       //assert (rt->rt_flags == RTF_UP);
       assert(rq->rq_dst == rt->rt_dst);
       //assert ((rt->rt_seqno%2) == 0);     // is the seqno even?
       sendReply_ic(rq->rq_src,
                 rt->rt_hops + 1,
                 rq->rq_dst,
                 rt->rt_seqno,
                 (u_int32_t) (rt->rt_expire - CURRENT_TIME),
                 //             rt->rt_expire - CURRENT_TIME,
                 rq->rq_timestamp,
                 NETWORK_DIAMETER,
                 delay_sec);
       // Insert nexthops to RREQ source and RREQ destination in the
      // precursor lists of destination and source respectively
      rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source
      rt0->pc_insert(rt->rt_nexthop); // nexthop to RREQ destination

       Packet::free(p);
     }
     else{
       /* shakal
        * Each sensor must confirm from whom the request packet has been sent
        * if it came from another cluster it is discarded.
        * if it is addressed to another cluster, it must be transformed into a EC request
        * otherwise it is forwarded (through common sensors)
        */
       // checking if I`m the cluster head
       bool sou_CH = (index%2048)==0;
       // checking if the request is addressed to another cluster
       bool eh_pra_fora = (floor(index/2048))!=(floor(rq->rq_dst/2048));

       if (sou_CH){ // I'm the cluster head!
         if (rq->rq_dst == index){ // is it addressed to me?
           if (last_rreq_ec.existRequest_ec(rq->rq_src)){ // Did I send a RRIni before?
             seqno = max(seqno, rq->rq_dst_seqno)+1;
             if (seqno%2) seqno++;
             sendReply_ic(rq->rq_src,           // IP Destination
                          1,                    // Hop Count
                          index,                // Dest IP Address
                          seqno,                // Dest Sequence Num
                          MY_ROUTE_TIMEOUT,     // Lifetime
                          rq->rq_timestamp,
                          NETWORK_DIAMETER,
```

```
                        aux);    // timestamp
            // get the stored RREQ_EC
            delay_sec += aux;
            rq_ec = HDR_AODV_REQUEST_EC(last_rreq_ec.getRequest_ec(rq->rq_src));
            aodv_rt_entry *rt1;
            rt1 = rtable.rt_lookup(rq->rq_src);
            if (rt1 && (rt1->rt_hops != INFINITY2) && (rt1->rt_seqno >= rq_ec->rq_dst_seqno)
) {
              assert(rq_ec->rq_dst == rt1->rt_dst);
              sendReply_ec(rq_ec->rq_src,
                           rt1->rt_hops + 1,
                           rq_ec->rq_dst,
                           rt1->rt_seqno,
                           (u_int32_t) (rt1->rt_expire - CURRENT_TIME),
                           rq_ec->rq_timestamp,
                           ih->ttl_,
                           delay_sec);
              Packet::free(p);
            }
           }else{ // I didn't send any RRIni before, so I'll just answer this request
              seqno = max(seqno, rq->rq_dst_seqno)+1;
              if (seqno%2) seqno++;
              sendReply_ic(rq->rq_src,           // IP Destination
                           1,                    // Hop Count
                           index,                // Dest IP Address
                           seqno,                // Dest Sequence Num
                           MY_ROUTE_TIMEOUT,     // Lifetime
                           rq->rq_timestamp,     // timestamp
                           NETWORK_DIAMETER,
                           delay_sec);
              Packet::free(p);
              return;
            }

          }else{ // it's not addressed to me...
            if (eh_pra_fora){ // is it addressed to another cluster?
              sendRequest_ec(rq->rq_dst,p,ih->ttl_,delay_sec);
            }else{ // I'm sorry, but the cluster head doesn't treat rreq_ic's...
              // release the packet...
              Packet::free(p);
              return;
            }
          }
        }else{ // I'm a common sensor (not CH)
           // forward the packet
           ih->saddr() = index;
           ih->daddr() = IP_BROADCAST;
           rq->rq_hop_count += 1;
           // Maximum sequence number seen en route
           if (rt) rq->rq_dst_seqno = max(rt->rt_seqno, rq->rq_dst_seqno);
           delay_sec +=DELAY;
           forward((aodv_rt_entry*) 0, p,delay_sec);
        }
      }
}

void
AODV::recvRequest_ec(Packet *p) {
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_request_ec *rq = HDR_AODV_REQUEST_EC(p);
aodv_rt_entry *rt;
char digest[20];
char dgst[20];
double delay_sec = hold_until - CURRENT_TIME;
if(delay_sec < 0.0) delay_sec = 0.0;

// checking if I`m the cluster head
 bool sou_CH = (index%2048)==0;
 // checking if the request is addressed to this cluster
 bool eh_pra_dentro = (floor(index/2048))==(floor(rq->rq_dst/2048));

 if (!sou_CH){
   // Only the CH can treat a RREQ_EC. Any other sensor that receive this kind of request
   // has to drop it off
   Packet::free(p);
   return;
 }
```

```
 if (sou_CH){ // I'm the CH, so I have to treat EC requests
   if (index == rq->rq_dst){ // Is it addressed to me?
     // Just to be safe, I use the max. Somebody may have
     // incremented the dst seqno.
     seqno = max(seqno, rq->rq_dst_seqno)+1;
     if (seqno%2) seqno++;
       bool valid = is_signature_valid_ec(p,&delay_sec);
       if(!valid) printf("signagure not valid\n");
       sendReply_ec(rq->rq_src,           // IP Destination
                    1,                    // Hop Count
                    index,                // Dest IP Address
                    seqno,                // Dest Sequence Num
                    MY_ROUTE_TIMEOUT,     // Lifetime
                    rq->rq_timestamp,
                    NETWORK_DIAMETER,
                    delay_sec);    // timestamp

     Packet::free(p);
   }
   else if (eh_pra_dentro){ // The request is addressed to my cluster!
     last_rreq_ec.addRequest_ec(rq->rq_dst,p);
     calculate_HASH_ec(p,&delay_sec,digest);
     calculate_TOP_HASH_ec(digest,(int)(NETWORK_DIAMETER - rq->rq_hop_count),&delay_sec,dgst);
     if(!is_signature_valid_ec(p,&delay_sec))
       printf("Invalid Signature");
     sendRRIni(rq->rq_dst,delay_sec);
     // calculate the time needed to perform cryptographic functions
     hold_until = CURRENT_TIME;
     hold_until += delay_sec;
     double aux = 0.0;
     calculate_DH_KEY_ec(rq->rq_public_key,&aux);
   }
   else{  // The request is addressed to another cluster!
     if (rt && (rt->rt_hops != INFINITY2) && (rt->rt_seqno >= rq->rq_dst_seqno) ) {

       //assert (rt->rt_flags == RTF_UP);
       assert(rq->rq_dst == rt->rt_dst);
       //assert ((rt->rt_seqno%2) == 0);     // is the seqno even?
       sendReply_ec(rq->rq_src,
                    rt->rt_hops + 1,
                    rq->rq_dst,
                    rt->rt_seqno,
                    (u_int32_t) (rt->rt_expire - CURRENT_TIME),
                 //             rt->rt_expire - CURRENT_TIME,
                    rq->rq_timestamp,
                    NETWORK_DIAMETER,
                    delay_sec);
       // Insert nexthops to RREQ source and RREQ destination in the
       // precursor lists of destination and source respectively
       rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source
       rt0->pc_insert(rt->rt_nexthop); // nexthop to RREQ destination
       // calculate the time needed to perform cryptographic functions
       hold_until = CURRENT_TIME;
       calculate_HASH_ec(p,&hold_until,digest);
       calculate_TOP_HASH_ec(digest,(int)(NETWORK_DIAMETER - rq-
>rq_hop_count),&hold_until,dgst);
       if(!is_signature_valid_ec(p,&hold_until))
         printf("Invalid Signature");
       Packet::free(p);
     }
     else {  //Can't reply. So forward the  Route Request
       ih->saddr() = index;
       ih->daddr() = IP_BROADCAST;
       rq->rq_hop_count += 1;
       // Maximum sequence number seen en route
       calculate_HASH_over_HASH_ec(rq->rq_hash,&delay_sec,rq->rq_hash);
       if (rt) rq->rq_dst_seqno = max(rt->rt_seqno, rq->rq_dst_seqno);
       delay_sec += DELAY;
       forward((aodv_rt_entry*) 0, p, delay_sec);
       // calculate the time needed to perform cryptographic functions
       hold_until = CURRENT_TIME;
       hold_until += delay_sec;
       calculate_HASH_ec(p,&hold_until,digest);
       calculate_TOP_HASH_ec(digest,(int)(NETWORK_DIAMETER - rq-
>rq_hop_count),&hold_until,dgst);
       if(!is_signature_valid_ec(p,&hold_until))
         printf("Invalid Signature");
```

```
     }
    }
 }
}

void
AODV::recvReply_ic(Packet *p) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply_ic *rp = HDR_AODV_REPLY_IC(p);
aodv_rt_entry *rt;
double pkt_size = ch->size() - IP_HDR_LEN; //size of data packet (CBR)
double aux;
char suppress_reply = 0;
double delay_sec = 0.0;
/*
 *   Security Management
 */
 if(!is_signature_valid_ic(p,&delay_sec)){  // if the signature is wrong, discard the packet
   Packet::free(p);                       // else, delay_sec contains the necessary time
   return;                              // to perform the cryptographic functions
 }
     aux = pkt_size / 64;
     aux = (double)ceil(aux);
     aux = aux * 0.000413; //RC5 encryption
     delay_sec += aux;

      forward(rt, buf_pkt, delay_sec);
      delay_sec += ARP_DELAY;
    }
  }
 }
 else {
  suppress_reply = 1;
 }

 /*
  * If reply is for me, discard it.
  */

if(ih->daddr() == index || suppress_reply) {
   Packet::free(p);
 }
 /*
  * Otherwise, forward the Route Reply.
  */
 else {
 // Find the rt entry
aodv_rt_entry *rt0 = rtable.rt_lookup(ih->daddr());
   // If the rt is up, forward
   if(rt0 && (rt0->rt_hops != INFINITY2)) {
        assert (rt0->rt_flags == RTF_UP);
     rp->rp_hop_count += 1;
     rp->rp_src = index;
     forward(rt0, p, delay_sec);
     // Insert the nexthop towards the RREQ source to
     // the precursor list of the RREQ destination
     rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source

   }
   else {
   // I don't know how to forward .. drop the reply.
#ifdef DEBUG
     fprintf(stderr, "%s: dropping Route Reply\n", __FUNCTION__);
#endif // DEBUG
     drop(p, DROP_RTR_NO_ROUTE);
   }
 }
}

void
AODV::recvReply_ec(Packet *p) {
//struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply_ec *rp = HDR_AODV_REPLY_EC(p);
aodv_rt_entry *rt;
char suppress_reply = 0;
```

```
char digest[20];
char dgst[20];
double delay_sec = hold_until - CURRENT_TIME;
double delay_aux = 0.0;
if(delay_sec < 0.0) delay_sec = 0.0;

// checking if I`m the cluster head
 bool sou_CH = (index%2048)==0;
 // checking if the request is addressed to another cluster
 bool eh_pra_dentro = (floor(index/2048))==(floor(ih->daddr()/2048));

 if (!sou_CH){
   // I'm not the CH, so discarding the packet...
   Packet::free(p);
   return;
 }

 /*
  * If reply is for me, discard it.
  */

if(ih->daddr() == index || suppress_reply) {
   Packet::free(p);
 }
 /*
  * Otherwise, forward the Route Reply.
  */
 else {
   aodv_rt_entry *rt0 = rtable.rt_lookup(ih->daddr());
   // If the rt is up, forward
   if(rt0 && (rt0->rt_hops != INFINITY2)) {
     if (eh_pra_dentro){

       // transforming the RREP_EC into RREP_IC
       calculate_HASH_ec(p,&delay_aux,digest);
       calculate_TOP_HASH_ec(digest,(int)(NETWORK_DIAMETER - rp-
>rp_hop_count),&delay_aux,dgst);
       if(!is_signature_valid_ec(p,&delay_aux))
         printf("Invalid Signature");
       sendReply_ic(ih->daddr(),
                    rp->rp_hop_count + 1,
                    rp->rp_dst,
                    rp->rp_dst_seqno,
                    (u_int32_t)rp->rp_lifetime,
                    rp->rp_timestamp,
                    ih->ttl_,
                    delay_aux);
       hold_until = CURRENT_TIME;
       hold_until +=delay_aux;
       calculate_DH_KEY_ec(rp->rp_public_key,&hold_until);
       Packet::free(p);
     }else{
 // Find the rt entry

       assert (rt0->rt_flags == RTF_UP);
       rp->rp_hop_count += 1;
       rp->rp_src = index;
       calculate_HASH_over_HASH_ec(rp->rp_hash,&delay_sec,rp->rp_hash);
       forward(rt0, p, delay_sec);
       // Insert the nexthop towards the RREQ source to
       // the precursor list of the RREQ destination
       rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source

     }

    }else {
     // I don't know how to forward .. drop the reply.
#ifdef DEBUG
       fprintf(stderr, "%s: dropping Route Reply\n", __FUNCTION__);
#endif // DEBUG
       drop(p, DROP_RTR_NO_ROUTE);
     }

 }
}
```

```
void
AODV::recvRRIni(Packet *p) {
struct hdr_aodv_rrini *rri = HDR_AODV_RRINI(p);
double delay_sec = 0.0;
/*
 * Drop the packet if I'm not the destination node
 * (The Cluster head can reach all nodes in his cluster, including the destination sensor)
 */

  if(rri->rri_dst != index) {
#ifdef DEBUG
    fprintf(stderr, "%s: this RRINI is not for me\n", __FUNCTION__);
#endif // DEBUG
    Packet::free(p);
    return;
  }
  else{ // I`m the destination node so I must establish a route to the cluster head
    if(is_signature_valid_rrini(p,&delay_sec));
    sendRequest_ic(rri->rri_src, delay_sec);
    Packet::free(p);
    return;
  }
}




/*
   Packet Transmission Routines
*/

void
AODV::forward(aodv_rt_entry *rt, Packet *p, double delay) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
double delay_sec = hold_until - CURRENT_TIME;
if(delay_sec < 0.0) delay_sec = 0.0;
delay_sec += delay;
double pkt_size = ch->size() - IP_HDR_LEN; //size of data packet (CBR)
double aux;
double ch_power = 1.0;
if (ch->ptype() != PT_AODV){
  if((index%2048)==0) ch_power = CH_POWER;
  aux = pkt_size / 16;
  aux = (double)ceil(aux);
  aux = aux * 0.000409; //RC5 decryption
  aux = aux / ch_power;
  delay_sec += aux;
}

 if(ih->ttl_ == 0) {

#ifdef DEBUG
  fprintf(stderr, "%s: calling drop()\n", __PRETTY_FUNCTION__);
#endif // DEBUG

  drop(p, DROP_RTR_TTL);
  return;
 }

 if (ch->ptype() != PT_AODV && ch->direction() == hdr_cmn::UP &&
       ((u_int32_t)ih->daddr() == IP_BROADCAST)
              || ((u_int32_t)ih->daddr() == (u_int32_t)here_.addr_)) {
   dmux_->recv(p,0);
   return;
 }

 if (rt) {
   assert(rt->rt_flags == RTF_UP);
   rt->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
   ch->next_hop_ = rt->rt_nexthop;
   ch->addr_type() = NS_AF_INET;
   ch->direction() = hdr_cmn::DOWN;       //important: change the packet's direction
 }
 else { // if it is a broadcast packet
   // assert(ch->ptype() == PT_AODV); // maybe a diff pkt type like gaf
   assert(ih->daddr() == (nsaddr_t) IP_BROADCAST);
```

```
   ch->addr_type() = NS_AF_NONE;
   ch->direction() = hdr_cmn::DOWN;       //important: change the packet's direction
 }

if (ih->daddr() == (nsaddr_t) IP_BROADCAST) {
 // If it is a broadcast packet
   assert(rt == 0);
   /*
    *  Jitter the sending of broadcast packets by 10ms
    */
   Scheduler::instance().schedule(target_, p,
                                  0.01 * Random::uniform());
 }
 else { // Not a broadcast packet
   if(delay > 0.0) {
     Scheduler::instance().schedule(target_, p, delay);
   }
   else {
   // Not a broadcast packet, no delay, send immediately
     Scheduler::instance().schedule(target_, p, 0.);
   }
 }

}


void
AODV::sendRequest_ic(nsaddr_t dst,double delay_sec) {
// Allocate a RREQ packet
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_request_ic *rq = HDR_AODV_REQUEST_IC(p);
char digest[20];
aodv_rt_entry *rt = rtable.rt_lookup(dst);
if(rt == 0) {
         rt = rtable.rt_add(dst);
 }
 assert(rt);

 /*
  *  Rate limit sending of Route Requests. We are very conservative
  *  about sending out route requests.
  */

 if (rt->rt_flags == RTF_UP) {
   assert(rt->rt_hops != INFINITY2);
   Packet::free((Packet *)p);
   return;
 }

 if (rt->rt_req_timeout > CURRENT_TIME) {
   Packet::free((Packet *)p);
   return;
 }

 // rt_req_cnt is the no. of times we did network-wide broadcast
 // RREQ_RETRIES is the maximum number we will allow before
 // going to a long timeout.

 if (rt->rt_req_cnt > RREQ_RETRIES) {
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
   rt->rt_req_cnt = 0;
 Packet *buf_pkt;
   while ((buf_pkt = rqueue.deque(rt->rt_dst))) {
       drop(buf_pkt, DROP_RTR_NO_ROUTE);
   }
   Packet::free((Packet *)p);
   return;
 }

#ifdef DEBUG
   fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d\n",
                    ++route_request, index, rt->rt_dst);
#endif // DEBUG

 // Determine the TTL to be used this time.
```

```
 // Dynamic TTL evaluation - SRD

 rt->rt_req_last_ttl = max(rt->rt_req_last_ttl,rt->rt_last_hop_count);

 if (0 == rt->rt_req_last_ttl) {
 // first time query broadcast
   ih->ttl_ = TTL_START;
 }
 else {
 // Expanding ring search.
   if (rt->rt_req_last_ttl < TTL_THRESHOLD)
     ih->ttl_ = rt->rt_req_last_ttl + TTL_INCREMENT;
   else {
   // network-wide broadcast
     ih->ttl_ = NETWORK_DIAMETER;
     rt->rt_req_cnt += 1;
   }
 }

 // remember the TTL used  for the next time
 rt->rt_req_last_ttl = ih->ttl_;

 // PerHopTime is the roundtrip time per hop for route requests.
 // The factor 2.0 is just to be safe .. SRD 5/22/99
 // Also note that we are making timeouts to be larger if we have
 // done network wide broadcast before.

 rt->rt_req_timeout = 2.0 * (double) ih->ttl_ * PerHopTime(rt);
 if (rt->rt_req_cnt > 0)
   rt->rt_req_timeout *= rt->rt_req_cnt;
 rt->rt_req_timeout += CURRENT_TIME;

 // Don't let the timeout to be too large, however .. SRD 6/8/99
 if (rt->rt_req_timeout > CURRENT_TIME + MAX_RREQ_TIMEOUT)
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
 rt->rt_expire = 0;

#ifdef DEBUG
 fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d, tout %f ms\n",
                 ++route_request,
                index, rt->rt_dst,
                rt->rt_req_timeout - CURRENT_TIME);
#endif  // DEBUG


 // Fill out the RREQ packet
 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rq->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_NONE;
 ch->prev_hop_ = index;          // AODV hack

 ih->saddr() = index;
 ih->daddr() = IP_BROADCAST;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;

 // Fill up some more fields.
 rq->rq_type = AODVTYPE_RREQ_IC;
 rq->rq_hop_count = 1;
 rq->rq_bcast_id = bid++;
 rq->rq_dst = dst;
 rq->rq_dst_seqno = (rt ? rt->rt_seqno : 0);
 rq->rq_src = index;
 seqno += 2;
 assert ((seqno%2) == 0);
 rq->rq_src_seqno = seqno;
 rq->rq_timestamp = CURRENT_TIME;
 calculate_HASH_ic(p,&delay_sec,digest);
 do_sign_ic(p,digest,&delay_sec);

 Scheduler::instance().schedule(target_, p, delay_sec);

}
```

```
void
AODV::sendRequest_ec(nsaddr_t dst, Packet *pkt,int ttl,double delay_sec) {
// Allocate a RREQ packet
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_request_ec *rq = HDR_AODV_REQUEST_EC(p);
struct hdr_aodv_request_ic *rq_ic;
aodv_rt_entry *rt = rtable.rt_lookup(dst);
char digest[20];
if(rt == 0) {
         rt = rtable.rt_add(dst);
 }

 assert(rt);

 /*
  *  Rate limit sending of Route Requests. We are very conservative
  *  about sending out route requests.
  */

 if (rt->rt_flags == RTF_UP) {
   assert(rt->rt_hops != INFINITY2);
   Packet::free((Packet *)p);
   return;
 }

 if (rt->rt_req_timeout > CURRENT_TIME) {
   Packet::free((Packet *)p);
   return;
 }

 // rt_req_cnt is the no. of times we did network-wide broadcast
 // RREQ_RETRIES is the maximum number we will allow before
 // going to a long timeout.

 if (rt->rt_req_cnt > RREQ_RETRIES) {
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
   rt->rt_req_cnt = 0;
 Packet *buf_pkt;
   while ((buf_pkt = rqueue.deque(rt->rt_dst))) {
       drop(buf_pkt, DROP_RTR_NO_ROUTE);
   }
   Packet::free((Packet *)p);
   return;
 }

#ifdef DEBUG
   fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d\n",
                   ++route_request, index, rt->rt_dst);
#endif // DEBUG

 // Determine the TTL to be used this time.
 // Dynamic TTL evaluation - SRD

 rt->rt_req_last_ttl = max(rt->rt_req_last_ttl,rt->rt_last_hop_count);

 if (0 == rt->rt_req_last_ttl) {
 // first time query broadcast
   ih->ttl_ = ttl;//TTL_START;
 }
 else {
 // Expanding ring search.
   if (rt->rt_req_last_ttl < TTL_THRESHOLD)
     ih->ttl_ = ttl;//rt->rt_req_last_ttl + TTL_INCREMENT;
   else {
   // network-wide broadcast
     ih->ttl_ = ttl;//NETWORK_DIAMETER;
     rt->rt_req_cnt += 1;
   }
 }

 // remember the TTL used  for the next time
 rt->rt_req_last_ttl = ih->ttl_;

 // PerHopTime is the roundtrip time per hop for route requests.
```

```
 // The factor 2.0 is just to be safe .. SRD 5/22/99
 // Also note that we are making timeouts to be larger if we have
 // done network wide broadcast before.

 rt->rt_req_timeout = 2.0 * (double) ih->ttl_ * PerHopTime(rt);
 if (rt->rt_req_cnt > 0)
   rt->rt_req_timeout *= rt->rt_req_cnt;
 rt->rt_req_timeout += CURRENT_TIME;

 // Don't let the timeout to be too large, however .. SRD 6/8/99
 if (rt->rt_req_timeout > CURRENT_TIME + MAX_RREQ_TIMEOUT)
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
 rt->rt_expire = 0;

#ifdef DEBUG
 fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d, tout %f ms\n",
                 ++route_request,
                index, rt->rt_dst,
                rt->rt_req_timeout - CURRENT_TIME);
#endif  // DEBUG


    // Fill out the RREQ packet
    // ch->uid() = 0;
    ch->ptype() = PT_AODV;
    ch->size() = IP_HDR_LEN + rq->size();
    ch->iface() = -2;
    ch->error() = 0;
    ch->addr_type() = NS_AF_NONE;
    ch->prev_hop_ = index;          // AODV hack  //mudar

    ih->saddr() = index;
    ih->daddr() = IP_BROADCAST;
    ih->sport() = RT_PORT;
    ih->dport() = RT_PORT;

  if (pkt==0){ // new request from the CH
    rq->rq_type = AODVTYPE_RREQ_EC;
    rq->rq_hop_count = 1;
    rq->rq_bcast_id = bid++;
    rq->rq_dst = dst;
    rq->rq_dst_seqno = (rt ? rt->rt_seqno : 0);
    rq->rq_src = index;
    seqno += 2;
    assert ((seqno%2) == 0);
    rq->rq_src_seqno = seqno;
    rq->rq_timestamp = CURRENT_TIME;
 }else{ // this request is forwarded by the CH!
    rq_ic = HDR_AODV_REQUEST_IC(pkt);
    rq->rq_type = AODVTYPE_RREQ_EC;
    rq->rq_hop_count = rq_ic->rq_hop_count + 1;
    rq->rq_bcast_id = rq_ic->rq_bcast_id;
    rq->rq_dst = rq_ic->rq_dst;
    rq->rq_dst_seqno = rq_ic->rq_dst_seqno;
    rq->rq_src = rq_ic->rq_src;
    rq->rq_src_seqno = rq_ic->rq_src_seqno;
    rq->rq_timestamp = rq_ic->rq_timestamp;
 }
 calculate_HASH_ec(p,&delay_sec,digest);
 strcpy(rq->rq_hash,digest);
 calculate_TOP_HASH_ec(digest, (int)NETWORK_DIAMETER,&delay_sec,rq->rq_top_hash);
 do_sign_ec(p,digest,&delay_sec);

 Scheduler::instance().schedule(target_, p, delay_sec);
 Packet::free(pkt);
}


void
AODV::sendRRIni(nsaddr_t dst,double delay_sec) {
// Allocate a RREQ packet
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_rrini *rri = HDR_AODV_RRINI(p);
aodv_rt_entry *rt = rtable.rt_lookup(dst);
if(rt == 0) {
```

```
        rt = rtable.rt_add(dst);
 }
 assert(rt);

 /*
  *  Rate limit sending of Route Requests. We are very conservative
  *  about sending out route requests.
  */

 if (rt->rt_flags == RTF_UP) {
   assert(rt->rt_hops != INFINITY2);
   Packet::free((Packet *)p);
   return;
 }

 if (rt->rt_req_timeout > CURRENT_TIME) {
   Packet::free((Packet *)p);
   return;
 }

 // rt_req_cnt is the no. of times we did network-wide broadcast
 // RREQ_RETRIES is the maximum number we will allow before
 // going to a long timeout.

 if (rt->rt_req_cnt > RREQ_RETRIES) {
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
   rt->rt_req_cnt = 0;
 Packet *buf_pkt;
   while ((buf_pkt = rqueue.deque(rt->rt_dst))) {
       drop(buf_pkt, DROP_RTR_NO_ROUTE);
   }
   Packet::free((Packet *)p);
   return;
 }

#ifdef DEBUG
   fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d\n",
                   ++route_request, index, rt->rt_dst);
#endif // DEBUG

 // Determine the TTL to be used this time.
 // Dynamic TTL evaluation - SRD

 rt->rt_req_last_ttl = max(rt->rt_req_last_ttl,rt->rt_last_hop_count);

 if (0 == rt->rt_req_last_ttl) {
 // first time query broadcast
   ih->ttl_ = TTL_START;
 }
 else {
 // Expanding ring search.
   if (rt->rt_req_last_ttl < TTL_THRESHOLD)
     ih->ttl_ = rt->rt_req_last_ttl + TTL_INCREMENT;
   else {
   // network-wide broadcast
     ih->ttl_ = 2;
     rt->rt_req_cnt += 1;
   }
 }

 // remember the TTL used  for the next time
 rt->rt_req_last_ttl = ih->ttl_;

 // PerHopTime is the roundtrip time per hop for route requests.
 // The factor 2.0 is just to be safe .. SRD 5/22/99
 // Also note that we are making timeouts to be larger if we have
 // done network wide broadcast before.

 rt->rt_req_timeout = 2.0 * (double) ih->ttl_ * PerHopTime(rt);
 if (rt->rt_req_cnt > 0)
   rt->rt_req_timeout *= rt->rt_req_cnt;
 rt->rt_req_timeout += CURRENT_TIME;

 // Don't let the timeout to be too large, however .. SRD 6/8/99
 if (rt->rt_req_timeout > CURRENT_TIME + MAX_RREQ_TIMEOUT)
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
 rt->rt_expire = 0;
```

```
#ifdef DEBUG
 fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d, tout %f ms\n",
                 ++route_request,
                index, rt->rt_dst,
                rt->rt_req_timeout - CURRENT_TIME);
#endif  // DEBUG


 // Fill out the RREQ packet
 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rri->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_NONE;
 ch->prev_hop_ = index;          // AODV hack

 ih->saddr() = index;
 ih->daddr() = IP_BROADCAST;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;

 // Fill up some more fields.
 rri->rri_type = AODVTYPE_RRINI;
 rri->rri_bcast_id = bid++;
 rri->rri_dst = dst;
 rri->rri_src = index;
 seqno += 2;
 assert ((seqno%2) == 0);

 do_sign_rrini(p,&delay_sec);

 Scheduler::instance().schedule(target_, p,delay_sec);

}

void
AODV::sendReply_ic(nsaddr_t ipdst, u_int32_t hop_count, nsaddr_t rpdst,
                u_int32_t rpseq, u_int32_t lifetime, double timestamp,int ttl,double
delay_sec) {
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply_ic *rp = HDR_AODV_REPLY_IC(p);
aodv_rt_entry *rt = rtable.rt_lookup(ipdst);
char digest[20];

#ifdef DEBUG
fprintf(stderr, "sending Reply from %d at %.2f\n", index, Scheduler::instance().clock());
#endif // DEBUG
 assert(rt);

 rp->rp_type = AODVTYPE_RREP_IC;
 //rp->rp_flags = 0x00;
 rp->rp_hop_count = hop_count;
 rp->rp_dst = rpdst;
 rp->rp_dst_seqno = rpseq;
 rp->rp_src = index;
 rp->rp_lifetime = lifetime;
 rp->rp_timestamp = timestamp;

 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rp->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_INET;
 ch->next_hop_ = rt->rt_nexthop;
 ch->prev_hop_ = index;          // AODV hack
 ch->direction() = hdr_cmn::DOWN;

 ih->saddr() = index;
 ih->daddr() = ipdst;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;
 ih->ttl_ = ttl;
```

```
 calculate_HASH_ic(p,&delay_sec,digest);
 do_sign_ic(p,digest,&delay_sec);
 Scheduler::instance().schedule(target_, p, 0.);

}

void
AODV::sendReply_ec(nsaddr_t ipdst, u_int32_t hop_count, nsaddr_t rpdst,
                u_int32_t rpseq, u_int32_t lifetime, double timestamp,int ttl,double
delay_sec) {
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply_ec *rp = HDR_AODV_REPLY_EC(p);
aodv_rt_entry *rt = rtable.rt_lookup(ipdst);
char digest[20];

#ifdef DEBUG
fprintf(stderr, "sending Reply from %d at %.2f\n", index, Scheduler::instance().clock());
#endif // DEBUG
 assert(rt);

 rp->rp_type = AODVTYPE_RREP_EC;
 //rp->rp_flags = 0x00;
 rp->rp_hop_count = hop_count;
 rp->rp_dst = rpdst;
 rp->rp_dst_seqno = rpseq;
 rp->rp_src = index;
 rp->rp_lifetime = lifetime;
 rp->rp_timestamp = timestamp;

 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rp->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_INET;
 ch->next_hop_ = rt->rt_nexthop;
 ch->prev_hop_ = index;          // AODV hack
 ch->direction() = hdr_cmn::DOWN;

 ih->saddr() = index;
 ih->daddr() = ipdst;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;
 ih->ttl_ = ttl;

 calculate_HASH_ec(p,&delay_sec,digest);
 strcpy(rp->rp_hash,digest);
 calculate_TOP_HASH_ec(digest, (int)NETWORK_DIAMETER - hop_count,&delay_sec,rp->rp_top_hash);
 do_sign_ec(p,digest,&delay_sec);

 Scheduler::instance().schedule(target_, p, delay_sec);

}
/*
 * Security Management
 */
//IC - Functions
void
AODV::calculate_HASH_ic(Packet *p,double *delay,char digest[20]){
  struct hdr_aodv *ah = HDR_AODV(p);
  struct hdr_aodv_request_ic *rq;
  struct hdr_aodv_reply_ic *rp;
  double ch_power = 1.0;
  double aux;
  int size;
  unsigned int dgst_len = 0;
  char message[255];

  EVP_MD_CTX md_ctx;
  EVP_MD_CTX_init(&md_ctx);
  char digest[20];
  switch(ah->ah_type) {
    case AODVTYPE_RREQ_IC:
      rq = HDR_AODV_REQUEST_IC(p);
      size = 48;  //size = strlen(message);
```

```
      break;

    case AODVTYPE_RREP_IC:
      rp = HDR_AODV_REPLY_IC(p);
      size = 44;  //size = strlen(message);
      break;
  }
  if((index%2048)==0) ch_power = CH_POWER;
  aux = (double)size / 64;
  aux = (double)ceil(aux);
  aux = aux * 0.003636;
  aux = aux / ch_power;
  *delay += aux;   // acording to Ganesan, et al. page 154.
  EVP_DigestInit(&md_ctx, EVP_ecdsa());
  EVP_DigestUpdate(&md_ctx, (const void*)message, size);
  EVP_DigestFinal(&md_ctx, (unsigned char *)digest, &dgst_len);
  //return digest;
}

void
AODV::do_sign_ic(Packet *p,char digest[20], double *delay){
  unsigned char in[16],out[16]="",ivb[16],enc[32]="";
  double  texto_len = 20;
  static unsigned char vetor[16];
  unsigned char key[16];
  RC5_32_KEY sch;
  int i,j,blk_sz=16;
  for (i=0;i<16;i++) key[i] = group_key[i];
  for (i=0;i<16;i++) vetor[i] = group_key[i+16];
  struct hdr_aodv *ah = HDR_AODV(p);
  struct hdr_aodv_request_ic *rq;
  struct hdr_aodv_reply_ic *rp;
  double ch_power = 1.0;
  if((index%2048)==0) ch_power = CH_POWER;
  RC5_32_set_key(&sch,16,key,12);
  for (j=0;j<(floor(texto_len/blk_sz));j++){
    for (i=0;i<blk_sz;i++) in[i] = digest[(j*blk_sz)+i]; //copiando o texto a ser cifrado em
blocos de 16 bytes
    memcpy(ivb,&vetor,blk_sz);
    RC5_32_cbc_encrypt(in,out,blk_sz,&sch,&(ivb[0]),RC5_ENCRYPT);
    for (i=0;i<blk_sz;i++) enc[(j*blk_sz)+i] = out[i]; //copiando o bloco cifrado para um
array unico
  }
  for (i=0;i<blk_sz;i++) in[i] = char(0x00); //inicializando o array para o final com bits 0
  for(i=j*blk_sz;i<texto_len;i++)in[i%blk_sz] = digest[i];
  memcpy(ivb,&vetor,blk_sz);
  RC5_32_cbc_encrypt(in,out,blk_sz,&sch,&(ivb[0]),RC5_ENCRYPT);
  for (i=0;i<blk_sz;i++) enc[(j*blk_sz)+i] = out[i];
  switch(ah->ah_type) {
    case AODVTYPE_RREQ_IC:
      rq = HDR_AODV_REQUEST_IC(p);
      strcpy(rq->rq_signature,(char *)enc);
      break;

    case AODVTYPE_RREP_IC:
      rp = HDR_AODV_REPLY_IC(p);
      strcpy(rp->rp_signature,(char *)enc);
      break;
  }
  *delay += (2*0.000413)/ch_power;   // acording to Ganesan, et al. page 154.
}

bool
AODV::is_signature_valid_ic(Packet *p, double *delay){
  char digest[20];
  char in[16],out[16]="",enc[32],dec[32];
  unsigned char ivb[16];

  static unsigned char vetor[16];
  unsigned char key[16];
  RC5_32_KEY sch;
  int i,j,enc_len,blk_sz=16;
  for (i=0;i<16;i++) key[i] = group_key[i];
  for (i=0;i<16;i++) vetor[i] = group_key[i+16];
  struct hdr_aodv *ah = HDR_AODV(p);
  struct hdr_aodv_request_ic *rq;
  struct hdr_aodv_reply_ic *rp;
```

```
  double ch_power = 1.0;
  if((index%2048)==0) ch_power = CH_POWER;
  RC5_32_set_key(&sch,16,key,12);
  switch(ah->ah_type) {
    case AODVTYPE_RREQ_IC:
      rq = HDR_AODV_REQUEST_IC(p);
      strcpy(enc,rq->rq_signature);
      break;

    case AODVTYPE_RREP_IC:
      rp = HDR_AODV_REPLY_IC(p);
      strcpy(enc,rp->rp_signature);
      break;
  }
  enc_len = 32;
  for (j=0;j<(floor(enc_len/blk_sz));j++){
    for (i=0;i<blk_sz;i++) in[i] = enc[(j*blk_sz)+i];
    memcpy(ivb,&vetor,blk_sz);
    RC5_32_cbc_encrypt((unsigned char *)in,(unsigned char
*)out,blk_sz,&sch,&ivb[0],RC5_DECRYPT);
    for (i=0;i<blk_sz;i++) dec[(j*blk_sz)+i] = out[i];
  }
  *delay += (2*0.000409)/ch_power;   // acording to Ganesan, et al. page 154.
  calculate_HASH_ic(p,delay,digest);
  for(i=0;i<20;i++) if(digest[i] != dec[i])return true; // was false
  return true;
}


//EC - Functions
void
AODV::calculate_HASH_ec(Packet *p,double *delay,char digest[20]){
  struct hdr_aodv *ah = HDR_AODV(p);
  struct hdr_aodv_request_ec *rq;
  struct hdr_aodv_reply_ec *rp;
  double ch_power = 1.0;
  int size;
  char message[255];
  unsigned int dgst_len;
  EVP_MD_CTX md_ctx;
  EVP_MD_CTX_init(&md_ctx);
  char digest[20];
  switch(ah->ah_type) {
    case AODVTYPE_RREQ_EC:
      rq = HDR_AODV_REQUEST_EC(p);
      size = 88;  //size = strlen(message);
      break;

    case AODVTYPE_RREP_EC:
      rp = HDR_AODV_REPLY_EC(p);
      size = 86;  //size = strlen(message);
      break;
  }
  if((index%2048)==0) ch_power = CH_POWER;
  *delay += (ceil((double)size/64)*0.007777)/ch_power;   // acording to Ganesan, et al. page
154.
  EVP_DigestInit(&md_ctx, EVP_ecdsa());
  EVP_DigestUpdate(&md_ctx, (const void*)message, size);
  EVP_DigestFinal(&md_ctx, (unsigned char *)digest, &dgst_len);
  return digest;
}

void
AODV::calculate_HASH_hop_count_ec(Packet *p,double *delay,char digest[20]){
  double ch_power = 1.0;
  struct hdr_aodv *ah = HDR_AODV(p);
  struct hdr_aodv_request_ec *rq;
  struct hdr_aodv_reply_ec *rp;

  int size;
  unsigned int dgst_len;
  char message[255];
  EVP_MD_CTX md_ctx;
  EVP_MD_CTX_init(&md_ctx);
  char digest[20];
  switch(ah->ah_type) {
    case AODVTYPE_RREQ_EC:
```

```
      rq = HDR_AODV_REQUEST_EC(p);
      //strcpy(message,rq->rq_hop_count);
      size = 8;  //size = strlen(message);
      break;

    case AODVTYPE_RREP_EC:
      rp = HDR_AODV_REPLY_EC(p);
      //strcpy(message,rp->rp_hop_count);
      size = 8;  //size = strlen(message);
      break;
  }
  if((index%2048)==0) ch_power = CH_POWER;
  *delay += (0.003636)/ch_power;   // acording to Ganesan, et al. page 154.
  EVP_DigestInit(&md_ctx, EVP_ecdsa());
  EVP_DigestUpdate(&md_ctx, (const void*)message, size);
  EVP_DigestFinal(&md_ctx, (unsigned char *)digest, &dgst_len);
  //return digest;
}

void
AODV::calculate_HASH_over_HASH_ec(char in[20],double *delay,char out[20]){
  double ch_power = 1.0;
  char dgst[20];
  unsigned int dgst_len;
  strcpy(dgst,in);
  EVP_MD_CTX md_ctx;
  EVP_MD_CTX_init(&md_ctx);
  if((index%2048)==0) ch_power = CH_POWER;
  *delay += 0.003636/ch_power;   // acording to Ganesan, et al. page 154.
  EVP_DigestInit(&md_ctx, EVP_ecdsa());
  EVP_DigestUpdate(&md_ctx, (const void*)dgst, 20);
  EVP_DigestFinal(&md_ctx, (unsigned char *)out, &dgst_len);
  //return dgst;
}

void
AODV::calculate_TOP_HASH_ec(char in[20], int times, double *delay, char out[20]){
  double ch_power = 1.0;
  int i;
  char dgst[20];
  unsigned int dgst_len;
  strcpy(dgst,in);
  EVP_MD_CTX md_ctx;
  EVP_MD_CTX_init(&md_ctx);
  if((index%2048)==0) ch_power = CH_POWER;
  *delay += (0.003636 * (double)times)/ch_power;   // acording to Ganesan, et al. page 154.
  for (i=0;i<times;i++){
    EVP_DigestInit(&md_ctx, EVP_ecdsa());
    EVP_DigestUpdate(&md_ctx, (const void*)in, 20);
    EVP_DigestFinal(&md_ctx, (unsigned char *)out, &dgst_len);
    strcpy(dgst,out);
  }
  return dgst;
}

void
AODV::do_sign_ec(Packet *p, char digest[20], double *delay){
  struct hdr_aodv *ah = HDR_AODV(p);
  struct hdr_aodv_request_ec *rq;
  struct hdr_aodv_reply_ec *rp;
  double ch_power = 1.0;

  if((index%2048)==0) ch_power = CH_POWER;
  *delay += 6.88/ch_power;   // Acording to Blab and Zitterbart, page 11
  switch(ah->ah_type) {
    case AODVTYPE_RREQ_EC:
      rq = HDR_AODV_REQUEST_EC(p);
      rq->rq_signature = ECDSA_do_sign((unsigned char *)digest, 20, ec_key);
      break;

    case AODVTYPE_RREP_EC:
      rp = HDR_AODV_REPLY_EC(p);
      rp->rp_signature = ECDSA_do_sign((unsigned char *)digest, 20, ec_key);
      break;
  }

}
```

```
bool
AODV::is_signature_valid_ec(Packet *p, double *delay){
  double ch_power = 1.0;
  char digest[20];
  struct hdr_aodv *ah = HDR_AODV(p);
  struct hdr_aodv_request_ec *rq;
  struct hdr_aodv_reply_ec *rp;


  int ret;

  switch(ah->ah_type) {
    case AODVTYPE_RREQ_EC:
      rq = HDR_AODV_REQUEST_EC(p);
      ret = ECDSA_do_verify((unsigned char *)digest, 20, rq->rq_signature, rq->rq_public_key);
      break;

    case AODVTYPE_RREP_EC:
      rp = HDR_AODV_REPLY_EC(p);
      ret = ECDSA_do_verify((unsigned char *)digest, 20, rp->rp_signature, rp->rp_public_key);
      break;
  }

  if((index%2048)==0) ch_power = CH_POWER;
  *delay += 24.17/ch_power;   // Acording to Blab and Zitterbart, page 11
  calculate_HASH_ec(p,delay,digest);

  return true;//ret==1;
}

void
AODV::calculate_DH_KEY_ec(EC_KEY *public_key,double *delay){
  unsigned char *buf=NULL;
  int len,out;
  double ch_power = 1.0;

  if((index%2048)==0) ch_power = CH_POWER;
  *delay += 17.28/ch_power;   // Acording to Blab and Zitterbart, page 11
  len = KDF1_SHA1_len;
  buf=(unsigned char *)OPENSSL_malloc(len);
  out = 1; //shakal
  out=ECDH_compute_key(buf,len,EC_KEY_get0_public_key(public_key),ec_key,KDF1_SHA1);
  strcpy(dh_key,(char *)buf);

}


//RRINI
void
AODV::calculate_HASH_rrini(Packet *p,double *delay,char digest[20]){
  double ch_power = 1.0;
  struct hdr_aodv_rrini *rri = HDR_AODV_RRINI(p);
  int size;
  unsigned int dgst_len;
  char message[255];
  EVP_MD_CTX md_ctx;
  EVP_MD_CTX_init(&md_ctx);
  //char digest[20];
  strcpy(message,rri->rri_signature);
  size = 16; //48 bytes(rri packet) minus 32 (signature)
  EVP_DigestInit(&md_ctx, EVP_ecdsa());
  EVP_DigestUpdate(&md_ctx, (const void*)message, size);
  EVP_DigestFinal(&md_ctx, (unsigned char *)digest, &dgst_len);
  //return digest;
  if((index%2048)==0) ch_power = CH_POWER;
  *delay += 0.003636 / ch_power;   // acording to Ganesan, et al. page 154.

}

void
AODV::do_sign_rrini(Packet *p, double *delay){
  unsigned char in[16],out[16]="",ivb[16],enc[32]="";
  static unsigned char vetor[16];
  unsigned char key[16];
  char digest[20];
  calculate_HASH_rrini(p,delay,digest);
```

```
  unsigned long  texto_len = strlen(digest);
  RC5_32_KEY sch;
  int i,j,blk_sz=16;
  for (i=0;i<16;i++) key[i] = group_key[i];
  for (i=0;i<16;i++) vetor[i] = group_key[i+16];
  struct hdr_aodv_rrini *rri = HDR_AODV_RRINI(p);
  double ch_power = 1.0;
  if((index%2048)==0) ch_power = CH_POWER;
  RC5_32_set_key(&sch,16,key,12);
  for (j=0;j<(floor(texto_len/blk_sz));j++){
    for (i=0;i<blk_sz;i++) in[i] = digest[(j*blk_sz)+i]; //copiando o texto a ser cifrado em
blocos de 16 bytes
    memcpy(ivb,&vetor,blk_sz);
    RC5_32_cbc_encrypt(in,out,blk_sz,&sch,&(ivb[0]),RC5_ENCRYPT);
    for (i=0;i<blk_sz;i++) enc[(j*blk_sz)+i] = out[i]; //copiando o bloco cifrado para um
array unico
  }
  for (i=0;i<blk_sz;i++) in[i] = char(0x00); //inicializando o array para o final com bits 0
  for(i=j*blk_sz;i<(int)texto_len;i++)in[i%blk_sz] = digest[i];
  memcpy(ivb,&vetor,blk_sz);
  RC5_32_cbc_encrypt(in,out,blk_sz,&sch,&(ivb[0]),RC5_ENCRYPT);
  for (i=0;i<blk_sz;i++) enc[(j*blk_sz)+i] = out[i];
  strcpy(rri->rri_signature,(char *)enc);
  *delay += (2*0.000413)/ch_power;   // acording to Ganesan, et al. page 154.
}

bool
AODV::is_signature_valid_rrini(Packet *p, double *delay){
  unsigned char in[16],out[16]="",ivb[16],enc[32]="",dec[32]="";
  static unsigned char vetor[16];
  unsigned char key[16];
  RC5_32_KEY sch;
  int i,j,blk_sz=16;
  for (i=0;i<16;i++) key[i] = group_key[i];
  for (i=0;i<16;i++) vetor[i] = group_key[i+16];
  struct hdr_aodv_rrini *rri = HDR_AODV_RRINI(p);
  RC5_32_set_key(&sch,16,key,12);
  strcpy((char *)enc,rri->rri_signature);
  int texto_len = strlen((char *)enc);
  for (j=0;j<=(floor(texto_len/blk_sz));j++){
    for (i=0;i<blk_sz;i++) in[i] = enc[(j*blk_sz)+i];
    memcpy(ivb,&vetor,blk_sz);
    RC5_32_cbc_encrypt(in,out,blk_sz,&sch,&(ivb[0]),RC5_DECRYPT);
    for (i=0;i<blk_sz;i++) dec[(j*blk_sz)+i] = out[i];
  }
  char digest[20];
  double ch_power = 1.0;
  if((index%2048)==0) ch_power = CH_POWER;
  *delay += (2*0.000409);   // acording to Ganesan, et al. page 154.
  calculate_HASH_rrini(p,delay,digest);
  for(i=0;i<20;i++) if(digest[i] != dec[i])return true; // was false;
  return true;
}
}
```

Parte do conteúdo do arquivo **cmu-trace.cc**:

```
/* -*-  Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *      This product includes software developed by the Computer Systems
 *      Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * Ported from CMU/Monarch's code, appropriate copyright applies.
 * nov'98 -Padma.
 *
 * $Header: /nfs/jade/vint/CVSROOT/ns-2/trace/cmu-trace.cc,v 1.88 2005/10/08 19:16:16 tomh Exp
$
 */
void
CMUTrace::format_aodv(Packet *p, int offset)
{
        struct hdr_aodv *ah = HDR_AODV(p);
        struct hdr_aodv_request_ic *rq_ic = HDR_AODV_REQUEST_IC(p);
       struct hdr_aodv_request_ec *rq_ec = HDR_AODV_REQUEST_EC(p);
        struct hdr_aodv_reply_ic *rp_ic = HDR_AODV_REPLY_IC(p);
       struct hdr_aodv_reply_ec *rp_ec = HDR_AODV_REPLY_EC(p);


        switch(ah->ah_type) {
        case AODVTYPE_RREQ_IC:

               if (pt_->tagged()) {
                   sprintf(pt_->buffer() + offset,
                           "-aodv:t %x -aodv:h %d -aodv:b %d -aodv:d %d "
                           "-aodv:ds %d -aodv:s %d -aodv:ss %d "
                           "-aodv:c REQUEST_INTRACLUSTER ",
                           rq_ic->rq_type,
                             rq_ic->rq_hop_count,
                             rq_ic->rq_bcast_id,
                             rq_ic->rq_dst,
                             rq_ic->rq_dst_seqno,
                             rq_ic->rq_src,
                             rq_ic->rq_src_seqno);
               } else if (newtrace_) {

                   sprintf(pt_->buffer() + offset,
                      "-P aodv -Pt 0x%x -Ph %d -Pb %d -Pd %d -Pds %d -Ps %d -Pss %d -Pc
REQUEST_INTRACLUSTER ",
                       rq_ic->rq_type,
                         rq_ic->rq_hop_count,
                         rq_ic->rq_bcast_id,
                         rq_ic->rq_dst,
                         rq_ic->rq_dst_seqno,
```

```
                        rq_ic->rq_src,
                        rq_ic->rq_src_seqno);


                } else {

                    sprintf(pt_->buffer() + offset,
                        "[0x%x %d %d [%d %d] [%d %d]] (REQUEST_INTRACLUSTER)",
                        rq_ic->rq_type,
                        rq_ic->rq_hop_count,
                        rq_ic->rq_bcast_id,
                        rq_ic->rq_dst,
                        rq_ic->rq_dst_seqno,
                        rq_ic->rq_src,
                        rq_ic->rq_src_seqno);
                }
                 break;

        case AODVTYPE_RREQ_EC:

                if (pt_->tagged()) {
                    sprintf(pt_->buffer() + offset,
                            "-aodv:t %x -aodv:h %d -aodv:b %d -aodv:d %d "
                            "-aodv:ds %d -aodv:s %d -aodv:ss %d "
                            "-aodv:c REQUEST_EXTRACLUSTER ",
                            rq_ec->rq_type,
                              rq_ec->rq_hop_count,
                              rq_ec->rq_bcast_id,
                              rq_ec->rq_dst,
                              rq_ec->rq_dst_seqno,
                              rq_ec->rq_src,
                              rq_ec->rq_src_seqno);
                } else if (newtrace_) {

                    sprintf(pt_->buffer() + offset,
                        "-P aodv -Pt 0x%x -Ph %d -Pb %d -Pd %d -Pds %d -Ps %d -Pss %d -Pc
REQUEST_EXTRACLUSTER ",
                        rq_ec->rq_type,
                          rq_ec->rq_hop_count,
                          rq_ec->rq_bcast_id,
                          rq_ec->rq_dst,
                          rq_ec->rq_dst_seqno,
                          rq_ec->rq_src,
                          rq_ec->rq_src_seqno);


                } else {

                    sprintf(pt_->buffer() + offset,
                        "[0x%x %d %d [%d %d] [%d %d]] (REQUEST_EXTRACLUSTER)",
                        rq_ec->rq_type,
                          rq_ec->rq_hop_count,
                          rq_ec->rq_bcast_id,
                          rq_ec->rq_dst,
                          rq_ec->rq_dst_seqno,
                          rq_ec->rq_src,
                          rq_ec->rq_src_seqno);
                }
                 break;

        // important to note that we are not interested in register HELLO e RERR messages
        case AODVTYPE_RREP_IC:
                if (pt_->tagged()) {
                    sprintf(pt_->buffer() + offset,
                            "-aodv:t %x -aodv:h %d -aodv:d %d -adov:ds %d "
                            "-aodv:l %f -aodv:c %s ",
                            rp_ic->rp_type,
                            rp_ic->rp_hop_count,
                            rp_ic->rp_dst,
                            rp_ic->rp_dst_seqno,
                            rp_ic->rp_lifetime,
                            "REPLY_INTRACLUSTER");
                } else if (newtrace_) {

                        sprintf(pt_->buffer() + offset,
                            "-P aodv -Pt 0x%x -Ph %d -Pd %d -Pds %d -Pl %f -Pc %s ",
                               rp_ic->rp_type,
```

```
                        rp_ic->rp_hop_count,
                        rp_ic->rp_dst,
                        rp_ic->rp_dst_seqno,
                        rp_ic->rp_lifetime,
                        "REPLY_INTRACLUSTER");
        } else {

                sprintf(pt_->buffer() + offset,
                        "[0x%x %d [%d %d] %f] (%s)",
                        rp_ic->rp_type,
                        rp_ic->rp_hop_count,
                        rp_ic->rp_dst,
                        rp_ic->rp_dst_seqno,
                        rp_ic->rp_lifetime,
                        "REPLY_INTRACLUSTER");
        }
         break;


// important to note that we are not interested in register HELLO e RERR messages
case AODVTYPE_RREP_EC:
        if (pt_->tagged()) {
            sprintf(pt_->buffer() + offset,
                    "-aodv:t %x -aodv:h %d -aodv:d %d -adov:ds %d "
                    "-aodv:l %f -aodv:c %s ",
                    rp_ec->rp_type,
                    rp_ec->rp_hop_count,
                    rp_ec->rp_dst,
                    rp_ec->rp_dst_seqno,
                    rp_ec->rp_lifetime,
                    "REPLY_EXTRACLUSTER");
        } else if (newtrace_) {

                sprintf(pt_->buffer() + offset,
                    "-P aodv -Pt 0x%x -Ph %d -Pd %d -Pds %d -Pl %f -Pc %s ",
                        rp_ec->rp_type,
                        rp_ec->rp_hop_count,
                        rp_ec->rp_dst,
                        rp_ec->rp_dst_seqno,
                        rp_ec->rp_lifetime,
                        "REPLY_EXTRACLUSTER");
        } else {

                sprintf(pt_->buffer() + offset,
                        "[0x%x %d [%d %d] %f] (%s)",
                        rp_ec->rp_type,
                        rp_ec->rp_hop_count,
                        rp_ec->rp_dst,
                        rp_ec->rp_dst_seqno,
                        rp_ec->rp_lifetime,
                        "REPLY_EXTRACLUSTER");
        }
         break;

case AODVTYPE_RRINI:
        if (pt_->tagged()) {
            sprintf(pt_->buffer() + offset,
                    "-aodv:t %x -aodv:h %d -aodv:d %d -adov:ds %d "
                    "-aodv:c %s ",
                    rp_ic->rp_type,
                    rp_ic->rp_hop_count,
                    rp_ic->rp_dst,
                    rp_ic->rp_dst_seqno,
                    "ROUTE_REQUEST_INITIATOR");
        } else if (newtrace_) {

                sprintf(pt_->buffer() + offset,
                    "-P aodv -Pt 0x%x -Ph %d -Pd %d -Pds %d -Pc %s ",
                        rp_ic->rp_type,
                        rp_ic->rp_hop_count,
                        rp_ic->rp_dst,
                        rp_ic->rp_dst_seqno,
                        "ROUTE_REQUEST_INITIATOR");
        } else {

                sprintf(pt_->buffer() + offset,
                        "[0x%x %d [%d %d]] (%s)",
```

```
                                rp_ic->rp_type,
                                rp_ic->rp_hop_count,
                                rp_ic->rp_dst,
                                rp_ic->rp_dst_seqno,
                                "ROUTE_REQUEST_INITIATOR");
                        }
                         break;

                default:
#ifdef WIN32
                         fprintf(stderr,
                                "CMUTrace::format_aodv: invalid AODV packet type\n");
#else
                        fprintf(stderr,
                                "%s: invalid AODV packet type\n", __FUNCTION__);
#endif
                         abort();
                }
}
```

ANEXO III

Arquivo de configuração da simulação para a interconexão entre três clusters: **003_clusters.tcl**

```
set val(chan)         Channel/WirelessChannel  ;# channel type
set val(prop)         Propagation/TwoRayGround ;# radio-propagation model
set val(ant)          Antenna/OmniAntenna      ;# Antenna type
set val(ll)           LL                       ;# Link layer type
set val(ifq)          Queue/DropTail/PriQueue  ;# Interface queue type
set val(ifqlen)       50                       ;# max packet in ifq
set val(netif)        Phy/WirelessPhy          ;# network interface type
set val(mac)          Mac/802_11               ;# MAC type
set val(nn)           23                        ;# number of mobilenodes
set val(rp)          AODV                     ;# routing protocol
set val(x)            620
set val(y)            20
set val(addType)      hierarchical

set ns [new Simulator]

set f [open 003_clusters.tr w]
$ns trace-all $f
set namtrace [open 003_clusters.nam w]
$ns namtrace-all-wireless $namtrace $val(x) $val(y)

set topo [new Topography]
$topo load_flatgrid $val(x) $val(y)

create-god $val(nn)

set chan_1 [new $val(chan)]
set chan_2 [new $val(chan)]


Phy/WirelessPhy set CPThresh_ 10.0
Phy/WirelessPhy set CSThresh_ 8.91e-17
Phy/WirelessPhy set RXThresh_ 1.92252e-13
Phy/WirelessPhy set Pt_ 2.818e-08


# CONFIGURE AND CREATE NODES

$ns node-config  -adhocRouting $val(rp) \
                 -llType $val(ll) \
                 -macType $val(mac) \
                 -ifqType $val(ifq) \
                 -ifqLen $val(ifqlen) \
                 -antType $val(ant) \
                 -propType $val(prop) \
                 -phyType $val(netif) \
                 -topoInstance $topo \
                 -agentTrace ON \
                 -routerTrace OFF \
                 -macTrace OFF \
                 -movementTrace OFF \
                 -channel $chan_1 \
                -addressType $val(addType)

AddrParams set domain_num_ 0           ;# number of domains
lappend cluster_num 1              ;# number of clusters in each domain
AddrParams set cluster_num_ $cluster_num
lappend eilastlevel 11 11 1         ;# number of nodes in each cluster
AddrParams set nodes_num_ $eilastlevel ;# of each domain

proc finish {} {
    global ns f namtrace
    $ns flush-trace
    close $f
    close $namtrace
    exit 0
}
```

```
set n(0) [$ns node 0.0.1]
$n(0) shape "circle"
$n(0) set X_ 10
$n(0) set Y_ 10
$n(0) set Z_ 0

set n(1) [$ns node 0.0.2]
$n(1) shape "circle"
$n(1) set X_ 20
$n(1) set Y_ 10
$n(1) set Z_ 0

set n(2) [$ns node 0.0.3]
$n(2) shape "circle"
$n(2) set X_ 30
$n(2) set Y_ 10
$n(2) set Z_ 0

set n(3) [$ns node 0.0.4]
$n(3) shape "circle"
$n(3) set X_ 40
$n(3) set Y_ 10
$n(3) set Z_ 0

set n(4) [$ns node 0.0.5]
$n(4) shape "circle"
$n(4) set X_ 50
$n(4) set Y_ 10
$n(4) set Z_ 0

set n(5) [$ns node 0.0.6]
$n(5) shape "circle"
$n(5) set X_ 60
$n(5) set Y_ 10
$n(5) set Z_ 0

set n(6) [$ns node 0.0.7]
$n(6) shape "circle"
$n(6) set X_ 70
$n(6) set Y_ 10
$n(6) set Z_ 0

set n(7) [$ns node 0.0.8]
$n(7) shape "circle"
$n(7) set X_ 80
$n(7) set Y_ 10
$n(7) set Z_ 0

set n(8) [$ns node 0.0.9]
$n(8) shape "circle"
$n(8) set X_ 90
$n(8) set Y_ 10
$n(8) set Z_ 0

set n(9) [$ns node 0.0.10]
$n(9) shape "circle"
$n(9) set X_ 100
$n(9) set Y_ 10
$n(9) set Z_ 0

set n(12) [$ns node 0.1.1]
$n(12) shape "circle"
$n(12) set X_ 520
$n(12) set Y_ 10
$n(12) set Z_ 0

set n(13) [$ns node 0.1.2]
$n(13) shape "circle"
$n(13) set X_ 530
$n(13) set Y_ 10
$n(13) set Z_ 0

set n(14) [$ns node 0.1.3]
$n(14) shape "circle"
$n(14) set X_ 540
$n(14) set Y_ 10
```

```
$n(14) set Z_ 0

set n(15) [$ns node 0.1.4]
$n(15) shape "circle"
$n(15) set X_ 550
$n(15) set Y_ 10
$n(15) set Z_ 0

set n(16) [$ns node 0.1.5]
$n(16) shape "circle"
$n(16) set X_ 560
$n(16) set Y_ 10
$n(16) set Z_ 0

set n(17) [$ns node 0.1.6]
$n(17) shape "circle"
$n(17) set X_ 570
$n(17) set Y_ 10
$n(17) set Z_ 0

set n(18) [$ns node 0.1.7]
$n(18) shape "circle"
$n(18) set X_ 580
$n(18) set Y_ 10
$n(18) set Z_ 0

set n(19) [$ns node 0.1.8]
$n(19) shape "circle"
$n(19) set X_ 590
$n(19) set Y_ 10
$n(19) set Z_ 0

set n(20) [$ns node 0.1.9]
$n(20) shape "circle"
$n(20) set X_ 600
$n(20) set Y_ 10
$n(20) set Z_ 0

set n(21) [$ns node 0.1.10]
$n(21) shape "circle"
$n(21) set X_ 610
$n(21) set Y_ 10
$n(21) set Z_ 0


Phy/WirelessPhy set Pt_ 9.73276e-04
set phy Phy/WirelessPhy
$ns node-config -phyType $phy
set n(10) [$ns node 0.0.0]
$n(10) shape "circle"
$n(10) set X_ 110
$n(10) set Y_ 10
$n(10) set Z_ 0

set n(11) [$ns node 0.1.0]
$n(11) shape "circle"
$n(11) set X_ 510
$n(11) set Y_ 10
$n(11) set Z_ 0

set n(22) [$ns node 0.2.0]
$n(22) shape "circle"
$n(22) set X_ 310
$n(22) set Y_ 10
$n(22) set Z_ 0

proc attach-CBR-traffic { node sink size interval } {
   #Get an instance of the simulator
   set ns [Simulator instance]
   #Create a CBR  agent and attach it to the node
   set cbr [new Agent/CBR]
   $ns attach-agent $node $cbr
   $cbr set packetSize_ $size
   $cbr set interval_ $interval

   #Attach CBR source to sink;
   $ns connect $cbr $sink
```

```
   return $cbr
  }
set sink [new Agent/Null] ;#LossMonitor]
$ns attach-agent $n(21) $sink

set cbr [attach-CBR-traffic $n(0) $sink 512 100.0]

$ns at 0.0000000001 "$cbr start"
#$ns at 0.4999999 "debug 1"
$ns at 105.0 "$cbr stop"
$ns at 200.0 "finish"

puts "Start of simulation.."
$ns run
```

ANEXO IV

PERIÓDICOS E EVENTOS RELEVANTES

Neste trabalho, foram utilizados artigos e feitas leituras nos materiais encontrados nos seguintes periódicos escritos:

- ACM Mobile Networks and Applications;
- ACM SIGMOBILE – Mobile Computing and Communications Review;
- IEEE COMPUTER MAGAZINE;
- IEEE INFOCOM – The Conference on Computer Communications;

Outros artigos foram encontrados também em anais de eventos realizados em várias partes do mundo e em diversos anos. Para encontra-los, usou-se das ferramentas de pesquisa como o sítio na Internet da ACM, da IEEE e o SCHOOLAR.GOOGLE. Os principais artigos foram publicados nos eventos a seguir:

- ACM CCS – Conference on Computer and Communications Security;
- ACM International Conference on Wireless Sensor Networks and Applications;
- ACM Mobicom – The Annual International Conference on Mobile Computing and Networking;
- ACM MobiHoc –International Symposium on Mobile Ad Hoc Networking and Computing;
- APNOMS – Asia-Pacific Network Operations and Management Symposium;
- CHES – Workshop on Cryptographic Hardware and Embedded Systems;
- DSOM – Distributed Systems: Operations and Management Workshop;
- IEEE SECON – Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks;
- LANOMS – Latin American Network Operations and Management Symposium;
- SAINT – IEEE International Symposium on Applications.