

REDUCING THE SCOPE OF DENIAL OF SERVICE
ATTACKS IN QUALITY OF SERVICE ROUTING
NETWORKS

By

MUNIRUL ISLAM

Bachelor of Science

North South University

Dhaka, Bangladesh

1999



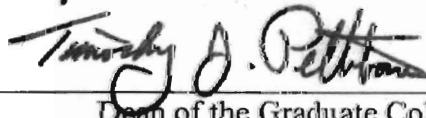
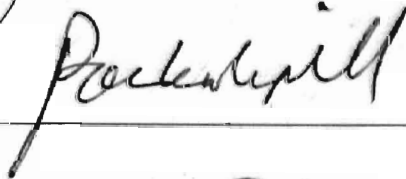
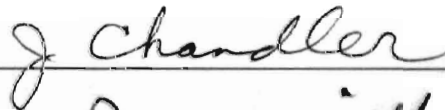
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2003

REDUCING THE SCOPE OF DENIAL OF SERVICE
ATTACKS IN QUALITY OF SERVICE ROUTING
NETWORKS

Thesis Approved:



Thesis Adviser



Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my major advisor, Dr. Johnson Thomas for his careful supervision, constructive guidance, precious suggestions, inspiration and friendship. My sincere gratitude extends to Dr. J. P. Chandler and Dr. Nohpill Park whose guidance, assistance, encouragement, and friendship are also invaluable.

I would also like to give my special appreciation to my mother, Mrs. Nasim Islam, for her love and encouragement. I wish to convey my gratefulness for her blessings and her support to inspire me to finish my degree. In addition my thanks extends to all my family members and friends who were there to motivate me throughout my research effort.

Finally, I would like to thank the members of the Department of Computer Science for their support during the two and a half years of my study.

TABLE OF CONTENTS

Chapter	Page
1. INTRODUCTION.....	1
1.1 MOTIVATION.....	2
2. LITERATURE REVIEW.....	4
2.1 DoS AND DDoS.....	4
2.1.1 SPOOFING.....	6
2.1.2 FILTER.....	6
2.2 ROUTING ON THE INTERNET.....	8
2.2.1 FIXED ROUTING.....	8
2.2.2 ADAPTIVE ROUTING.....	9
2.2.2.1 DISTANCE VECTOR ALGORITHM.....	9
2.2.2.2 LINK STATE ALGORITHM.....	11
2.3 QoS ROUTING.....	12
2.3.1 BELLMAN-FORD-MOORE ALGORITHM.....	13
2.3.2 BANDWIDTH-DELAY CONSTRAINED QoS ROUTING.....	17
3. THESIS OBJECTIVE.....	18
4. DESIGN.....	19
4.1 ASSUMPTION.....	19

4.2 INPUT.....	19
4.3 SECURE BELLMAN-FORD-MOORE ALGORITHM.....	20
4.3.1 A METRIC FOR SECURITY.....	21
4.3.2 MODIFIED SECURE BMF ALGORITHM.....	23
4.3.3 TRACEBACK.....	24
4.4 SECURE DISTANCE VECTOR ALGORITHM.....	24
4.4.1 DENIAL OF SERVICE ATTACK.....	27
4.4.2 TRACEBACK.....	28
5. IMPLEMENTAION APPROACH.....	29
5.1 PROTOCOL DESIGN.....	30
5.1.1 FINDING OUT NEIGHBORS.....	30
5.1.2 FINDING OUT FILTER COVERAGE FOR A NODE.....	30
5.2 SECURE ROUTING PROTOCOL BASED ON BELLMAN-FORD-MOORE ALGORITHM.....	31
5.2.1 PACKET FIELDS.....	31
5.2.2 PATH RESERVATION.....	39
5.2.3 TRACEBACK.....	40
5.3 SECURE ROUTING PROTOCOL BASED ON DISTANCE VECTOR ALGORITHM.....	41
5.3.1 PACKET FIELDS	41
6. SIMULATION RESULTS AND ANALYSIS.....	47

7. CONCLUSIONS.....	56
REFERENCES.....	57
APPENDICES.....	59
APPENPIX A – GLOSSARY.....	59
APPENDIX B – IMPLEMENTATION.....	60

LIST OF FIGURES

Figure	Page
1. DoS Attack.....	5
2. DdoS Attack.....	6
3. Input Example.....	20
4. Hello Packet Travel Path.....	25
5. Flow Diagram for Secure DVA.....	27
6. Neighbor Table Construction.....	30
7. Reservation of a Path.....	40
8. Security on 20-node network.....	48
9. Security on 40-node network	50
10. Security on 60-node network	52
11. Security on 80-node network	54

LIST OF TABLES

Table	Page
1. Forwarding Table for node 1	26

NOMENCLATURE

DoS	Denial of Service
DDoS	Distributed Denial of Service
QoS	Quality of Service
DVA	Distance Vector Algorithm
BFM	Bellman-Ford-Moore

Chapter 1

Introduction

Denials of Service (DoS) attacks consume the resources of a remote host or network that would otherwise be used for serving legitimate users. DoS continue to be a pervasive problem for Internet services, as evidenced by the recent denial of service at Computer Emergency Response Team (CERT) [7]. Such attacks require each attacking node to perform only a small amount of work, relying on the cumulative efforts of many machines to overload the victim node. Attacks range from simple Internet Control Message Protocol (ICMP) ping requests to sophisticated attacks that are difficult to distinguish from a sudden spike in legitimate use.

In a Distributed Denial of Service (DDoS) attack, the perpetrators may spend weeks or months subverting hundreds or thousands of machines by exploiting well-known security flaws. Once the machines are subverted, the perpetrator installs tools to execute an attack. On command, the prepared machines (known as “zombies”) collectively target a specified victim with a packet storm consisting of repeated packet requests. The packets may also have forged return addresses, allowing one machine to generate requests that appear to be coming from hundreds of machines, and making the sources of the attack difficult to trace. This problem may intensify since newer operating systems that are being introduced may make it easier to spoof IP addresses [6]. Halting these attacks is nontrivial and typically relies on filtering at the router (or just waiting out the “packet storm”).

New multimedia applications and message passing demand more bandwidth and are less tolerant to jitters, delays and lost packets than the traditional data applications

that have so far dominated the networks. Real-time quality of service (QoS) of the network is essential for such applications on low capacity links and over long hauls.

Although the issues of QoS and resource reservation have been studied in great detail, but the QoS protocols do not offer any form of security. In this paper, we explore these issues by implementing two protocols that provide QoS with a measure of security for peer to peer networking. The first protocol reserves the best secured path for multimedia applications and the second protocol finds the most secured path for a message on an ad-hoc basic.

1.1 Motivation:

According to CNN, In February 2000 a number of popular websites were slowed or shut down by distributed denial of service attacks. Amazon.com, Buy.com, CNN.com, eBay, E*Trade and ZDNet were all victims of these attacks. Analysts estimated that Yahoo! lost \$500,000 in e-commerce and advertising revenue when it was knocked offline for three hours. [14]

Part of the problem with DoS attacks is the sheer number of ways in which they can operate. A DoS attack can be launched to overwhelm a target's Web site, CPU, memory, network bandwidth or routers. It can also work by taking advantage of known flaws in products. [5]

Distributed DoS (DDoS) attacks are another variation. Such assaults, which are more difficult to detect than other DoS attacks, involve short-lived bursts of spurious traffic directed at a target from multiple sources and are aimed at slowing network performance. The problem arises because almost all DoS attacks involve multiple

networks and attack sources, many of which have spoofed IP addresses to make detection even harder.

These alarming issues need to be addressed and plethora of design parameters must be considered in order to come up with a secure network system.

Chapter 2

Literature Review

2.1 DoS and DDoS:

DoS attacks, when successful, make a system difficult or impossible to use. Some attacks crashes the systems, others merely make them so busy that no useful work is done. There are two principal classes of attacks:

Logic attacks and flooding attacks. Attacks in the first class, such as the “Ping-of-Death”, exploit existing software flaws to cause remote servers to crash or substantially degrade in performance. Either upgrading faulty software or filtering particular packet sequences can prevent many of these attacks, but they remain a serious and ongoing threat. The second class, flooding attacks, overwhelm the victim’s CPU, memory, or network resources by sending large numbers of spurious requests. Because there is typically no simple way to distinguish the “good” requests from the “bad”, it can be extremely difficult to defend against flooding attacks.

The WWW Security FAQ [11] describes a DoS attack as:

“an attack designed to render a computer or network incapable of providing normal services. The most common DoS attacks will target the computer’s network bandwidth or connectivity. Bandwidth attacks flood the network with such a high volume of traffic, which all available network resources are consumed and legitimate user requests cannot get through. Connectivity attacks flood a computer with such a high volume of connection requests, that all available operating system resources are consumed, and computer can no longer process legitimate user requests.”

Figure 1 (below) depicts the scheme of a simple DoS attack. The services will be denied until the source of the attack can be identified and calls from source can be blocked. Here, the attacker takes control of a computer (zombie) which then become the source for the actual attack.

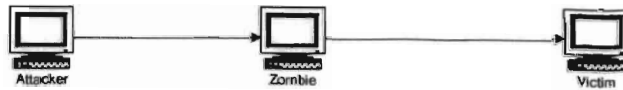


Figure 1: DoS attack [9]

Consequently attackers have discovered new ways of bringing these system to its knees. They don't use single hosts for their attacks but they also cluster several dozens or even hundreds of computers to do a coordinated strike. The WWW Security FAQ [11] on Distributed Denial of Service (DDoS) attacks, as these forms are called:

“A Distributed Denial of Service (DDoS) attack uses many computers to launch a coordinated DoS attack against one or more targets. Using client/server technology, the perpetrator is able to multiply the effectiveness of the Denial of Service significantly by harnessing the resources of multiple unwitting accomplice computers that serve as attack platforms.”

Figure 2 depicts a DDoS network where the attacker controls one or more computers (masters), which then control several more zombies. This in turn means more pathways that need to be blocked, along with a dramatic increase in the amount of resources being consumed by the target.

As we can see, there is only one attacker in a DDoS attack. The attacker takes over a number of masters. The masters control several zombies. And the zombies do the actual attack and bring down a network.

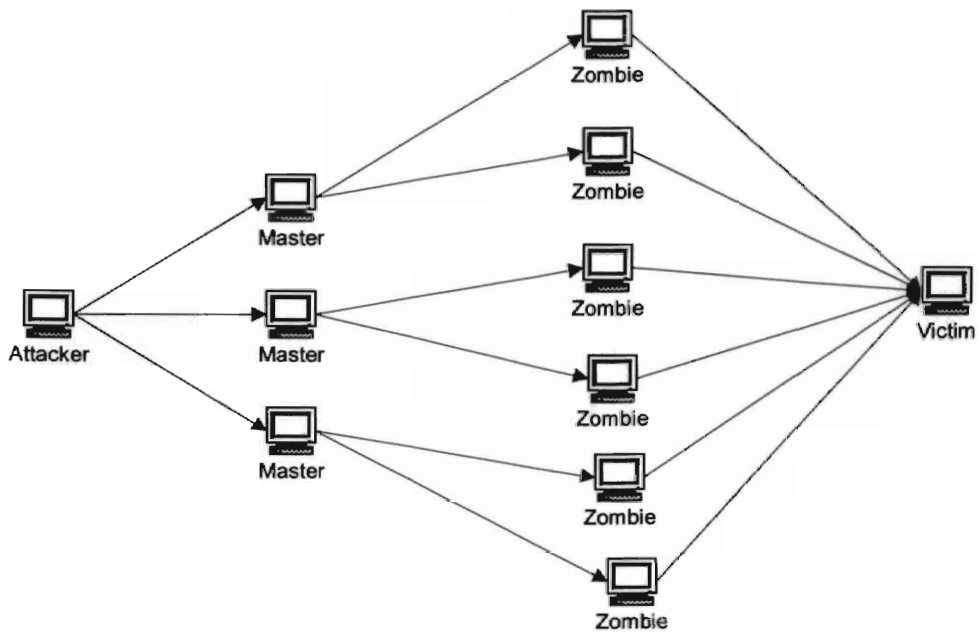


Figure 2: DDoS Attack [9]

2.1.1 Spoofing:

To conceal their location, thereby forestalling an effective response, attackers typically forge, or “spoof”, the source address of each packet they send. Consequently, the packets appear to the victim to be arriving from one or more third parties.

2.1.2 Filter:

Consider a point-to-point communication network representing as a directed graph $G = (V, E)$, where V is the set of all nodes and E is the set of links in G . Let $\zeta(u, v)$ denote the set of all loop-free paths from u to where $u, v \in V$. A routing algorithm and its computed routes lead to a subset $R(u, v) \subseteq \zeta(u, v)$. An IP packet $M(s, t)$ with source IP address s and destination IP address t is routed through the network according to $R(s, t)$.

A filter [4] $F_e: V^2$ is a function defined for the link $e = (u, v) \in E$ where this is interpreted to mean that a router in v acting as a peering inspects an IP packet $M(s, t)$

arriving on e , then decides whether to forward the packet ($F_e(s,t) = 0$), or filter-i.e., discard-the packet ($F_e(s,t) = 1$). We call F_e a route-based packet filter with respect to R if

$$F_e(s,t) = 0 \text{ for } e \in R(s,t).$$

With a slight abuse of notation, we can use “ $e \in R(s,t)$ ” to mean that link e is on some path belonging to $R(s,t)$. Thus a route-based filter is *safe* in the sense that it does not discard packet that are potentially consistent with respect to R as judged locally at e . A route-based filter is *maximal* [4] if it satisfies $F_e(s,t) = 0$ if, and only if, there exists a path in $R(s,t)$ with e as one of its links. Thus a maximal route-based filter carries out all the filtering of spoofed IP traffic that is possible without adversely affecting routing of non-spoofed IP packets as determined by R .

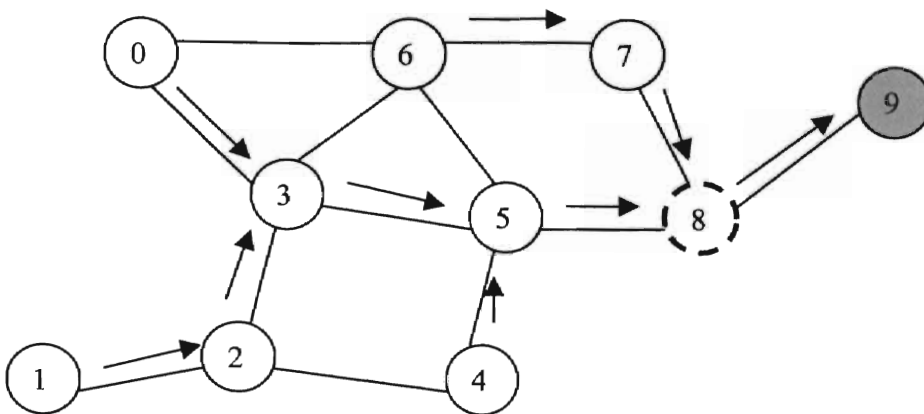
A *semi-maximal* [4] filter is a maximal filter which uses only the source IP address of a packet to out its filtering (i.e., a projection of F_e). In other words, $F_e(s,t)$ is a semi-maximal filter with respect to R if

$$F_e(s,t) = \begin{cases} 0, & \text{if } e \in R(s,v) \text{ for } v \in V; \\ 1, & \text{otherwise} \end{cases}$$

Hence, its filtering capability is, in general, less than that of its maximal counterpart. Although we lose in potential filtering power - but a semi-maximal filter can be represented by a filtering table in linear space, which brings it to the domain of feasibility, if not practicality.

We include semi-maximal filters to achieve security features within a network. In this paper, filter and semi-maximal filter notation is used interchangeably.

Example:



With packet filtering executed at node 8, the spoofable address range at attack site 1, is reduced from $S_{1,9} = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ to $\{0, 1, 2, 3, 4, 5\}$.

2.2 Routing on the Internet:

Internet[8] is a packet switched network where routing is done on a per packet basis. Devices called routers make the routing decisions. The routers are responsible for receiving and forwarding packets through the network. Each router maintains a routing table, which acts as reference to decide the next hop router to which a packet should be forwarded. Depending on how the routing tables are updated, routing can be classified into two categories: Fixed Routing and Adaptive Routing.

2.2.1 Fixed Routing:

Fixed routing is a simple routing scheme where a single permanent route is fixed for each source and destination pair in the network. The routes are mostly fixed, unless there is a change in the topology of the network. Hence, in this case the routing cannot be performed on any kind of dynamic variables such as cost, delay or bandwidth. Fixed

routing requires the routing tables to be pre-configured and so is not capable of taking any action during congestion or network failures.

2.2.2 Adaptive Routing:

As the name implies, adaptive routing is adaptive to changes in the network. The two main conditions that influence the routing decisions are: network failure and congestion.

Though adaptive routing sounds very attractive, it has its drawbacks too.

- Algorithms to determine paths based on dynamic parameters get very complex, increasing the processing burden on the routers.
- To make effective routing decisions, the routers have to be updated from time to time about the state of the network. But this update adds overhead to the already burdened network.

Adaptive routing algorithms can be classified based on the way a router updates its information. If the adjacent routers initiate the update it is called the Distance Vector Routing algorithm, and if the updates are based on information from all the routers it is called the Link State Algorithm.

2.2.2.1 Distance Vector Algorithm:

Distance vector algorithms [15] based on a table giving the best route to every destination in the system. We have to define a “metric” in order to find the best route. In simple networks, it is common to use a metric that simply counts how many gateways a message must go through. In more complex networks, a metric is chosen to represent the total amount of delay that the message suffers, the cost of sending it, or some other quantity, which may be minimized. The main requirement is that it must be possible to represent the metric as a sum of “costs” for individual hops.

Formally, if it is possible to get from node i to node j directly, then a cost, $d(i,j)$ is associated with the hop between i and j . In the normal case where all entities on a given network are considered to be the same, $d(i,j)$ is the same for all destinations on a given network, and represents the cost of using the network. To get the metric of a complete route, one just adds up the costs of individual hops that makes up the route. We assume that the costs are positive integers.

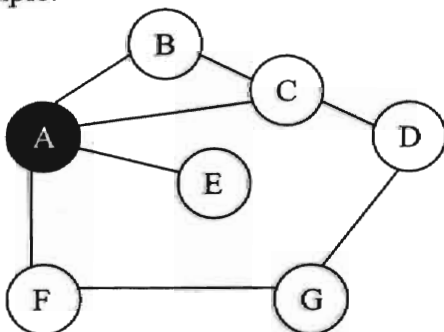
Let $d(i,j)$ represent the metric of the best route from node i to node j . It should be defined for every pair of nodes. $d(i,j)$ represents the costs of individual steps. Formally, let $d(i,j)$ represent the cost of going directly from node i to node j . It is infinite if i and j are not immediate neighbors. Here, $d(i,i)$ is infinite. So, there is no direct path from a node to itself. Since the costs are additive, it is easy to show that the best metric must be described by

$$d(i,j) = 0, \quad \text{all } i$$

$$d(i,j) = \min_k [d(i,k) + d(k,j)], \quad \text{otherwise}$$

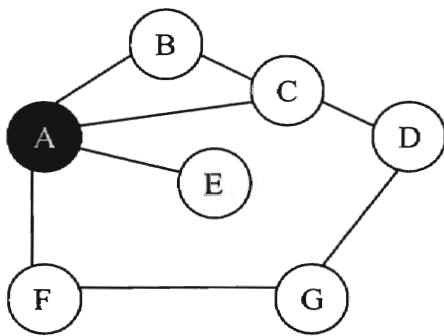
and that the best routes start by going from i to those neighbors k for which $d(i,k) + d(k,j)$ has the minimum value.

Example:



Dest	Cost	Next
B	1	B
C	1	C
D	∞	-
E	1	E
F	1	E
G	∞	-

A graph of 7 nodes. Initial cost vector at A. We are doing the routing from A's perspective. Initially A receives hello packets from its neighbors and updates its table accordingly.



Dest	Cost	Next
B	1	B
C	1	C
D	2	C
E	1	E
F	1	E
G	2	F

After the first iteration, A's table has information for all the nodes.

2.2.2.2 Link State Algorithm:

In these algorithms a router does not limit its updates only to its neighboring routers, rather the routing updates are flooded to all the routers in the network. Link state algorithms are so called because, instead of advertising a list of distances to each known destination, a router running a link state algorithm floods the network with state of the local link. The result is that all routers obtain the same database of collected advertisements, collectively describing the current map of the network. Using this

network map each router runs Dijkstra's algorithm [13] on it, resulting in shortest path distances to each of the destinations.

In Dijkstra's algorithm we start from source towards destination. Steps in Dijkstra's algorithm:

1. Set probe node to starting node.
2. Probe neighboring node and tentatively label them with (probe node, cumulative distance from start).
3. Search all tentatively labeled nodes for the minimum label, make this minimum node's label permanent, make it the new probe node.
4. If the probe node is the destination node, stop, else goto 2.

Here, the distance part of the node labels is cumulative distance from the source node, not simply distance from the last probe node.

2.3 QoS Routing:

Under QoS based routing, paths for flows would be determined based on some knowledge of resource availability in the network as well as the QoS requirements of flows. The metrics used to compute the paths play a very important role in QoS routing, since these metrics represent the basic network properties of interest, such as residual bandwidth, delay, jitter and monetary cost.

In our discussions we assume one metric on any link, i.e., bandwidth.

The QoS routing problems can be divided into two major classes: unicast routing and multicast routing. Unicast routing aims at finding paths from source to the destination satisfying one or more constraints. On the other hand, multicast routing aims at finding an optimal tree such that the paths from the source to a set of destinations

satisfy one or more constraints. Generally speaking, unicast routing could be considered as a special case of multicast routing problem. This thesis deals with unicast routing problem.

We first give an example of delay-constrained unicast routing. We do that by explaining the Bellman-Ford-Moore (BFM) algorithm.

2.3.1 The BFM algorithm:

Consider a point-to-point communication network representing as a directed graph $N = (V, E)$, where V is the set of all nodes and E is the set of links in N . The algorithm associates to variables with node u , namely $CLABEL(u)$ and $PRED(u)$. At any step in this algorithm $CLABEL(u)$ represents the cost of a path from node s to node u and $PRED(u)$ indicates the node from which node u has received its latest $CLABEL$ value. Initially $CLABEL(s)=0$, $CLABEL(u)=\infty$, for all $u \in V$ and $u \neq s$ and $PRED(u)=u$ for all $u \in V$.

The general step in the algorithm is:

Pick any link (u,v) such that $CLABEL(u) \neq \infty$ and $CLABEL(v) > CLABEL(u) + c_{u,v}$. Set $CLABEL(v) = CLABEL(u) + c_{u,v}$ and $PRED(v) = u$.

If no such link is available, the BFM algorithm terminates.

Here, t is the destination. At termination $CLABEL(t)$ gives the cost of a minimum cost $s-t$ path. This path can be traced starting at $PRED(t)$ and working backwards towards the source node s .

We next present an efficient $O(m,n)$ implementation of BFM algorithm [13].

In the following discussion scanning a node u means examining all the edges (u,v) and labeling the neighbor nodes of u , if possible. A sweep of the BFM algorithm refers to the process of scanning all the nodes $1,2,3,\dots,n$ in that order.

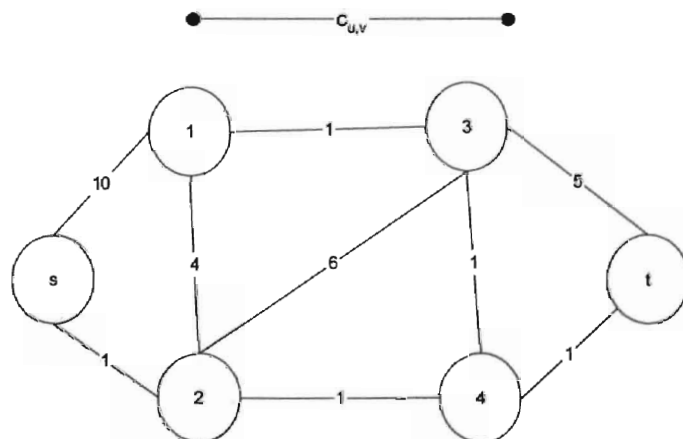
Following is an implementation of the BFM algorithm using the concept of the sweep.

The BFM algorithm:

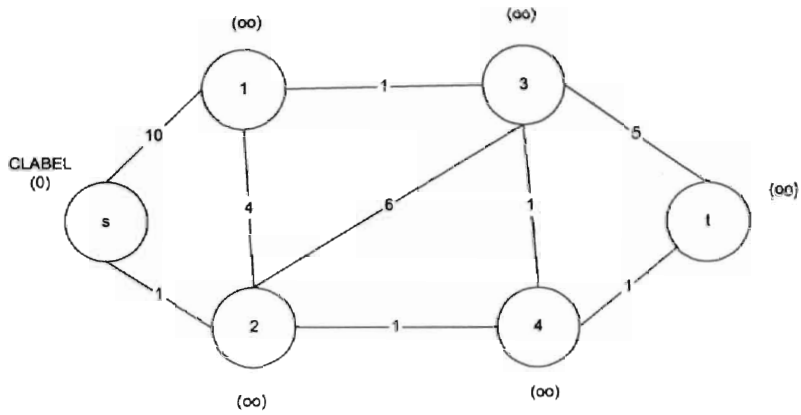
1. (INITIALIZATION) Set $CLABEL(s) = 0$, $CLABEL(u) = \infty$, for all $u \in V$ and $u \neq s$ and $PRED(u) = u$, for all $u \in V$.
2. Perform a Sweep.
3. If no node $CLABEL$ value gets updated, STOP. $CLABEL(t)$ gives the cost of the min-cost shortest s - t path. Otherwise, repeat step 2.

It can be shown that after performing n sweeps the BFM algorithm will terminate, resulting in the time complexity of $O(mn)$.

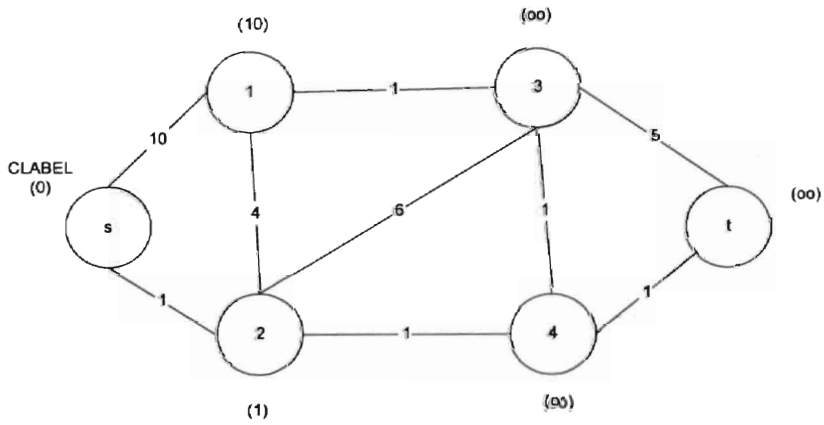
Here, while scanning a node u during a sweep, the current value of $CLABEL(u)$ is used to label the neighbors of node u .



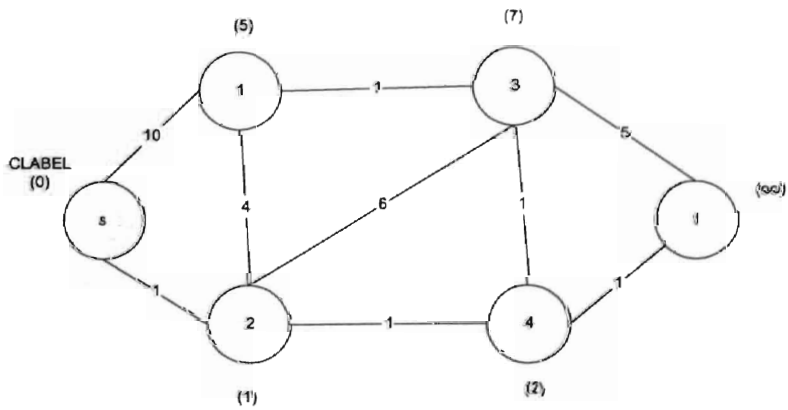
Initialization:



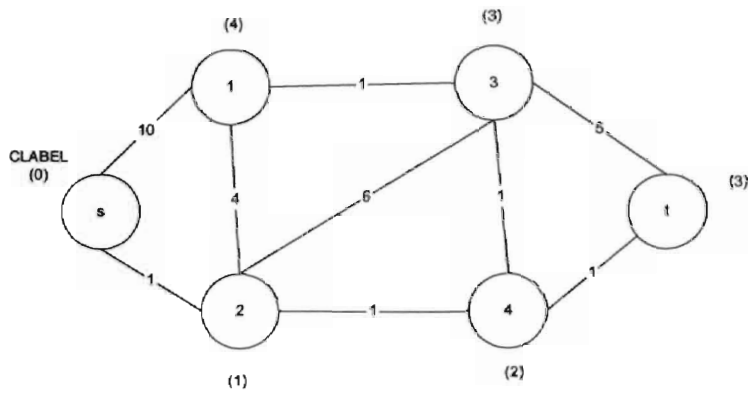
Sweep 1:



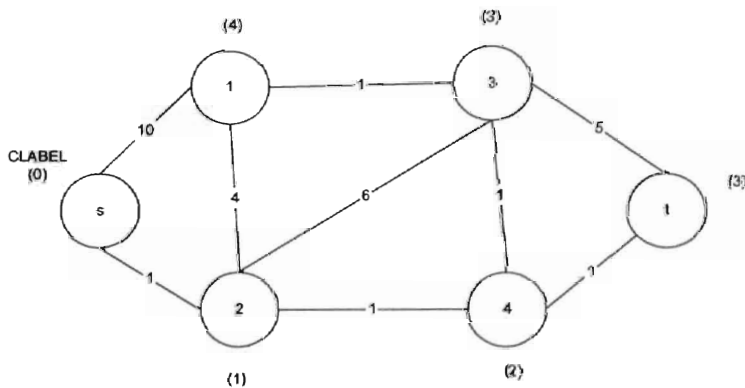
Sweep 2:



Sweep 3:



Sweep 4:



So, the path is s-2-4-t.

As we can see the BFM algorithm is only concerned with finding a QoS path from source to destination. It does not have any security features. To deal with situation, we have devised a QoS heuristic based on the BFM algorithm. To include security feature, we have use some of the nodes as filters. By using filters we are able to block some of the IP spoofing attacks. Using the filter information we can traceback to a set of nodes, which are capable of spoofing.

Next we talk about Bandwidth-Delay constrained QoS routing [16]. It is based on the standard Dijkstra's algorithm.

2.3.2 Bandwidth-Delay Constrained QoS routing:

The problem is defined as finding a path that has bandwidth greater than B and having total delay less than D . This algorithm first deletes all those nodes from the graph that has bandwidth less than B . From the remaining graph, we find the path with length less than threshold based on Dijkstra's algorithm. The algorithm is:

1. Set $d_{ij} = \infty$, if $b_{ij} < B$
2. Set $L = \{1\}$, $D_i = b_{1i}$ for all $i \neq 1$
3. Find $k \notin L$ so that $D_k = \min D_i$

If $D_k > D$, no such a path can be found and algorithm terminates.

If L contains node m , a path is found and algorithm terminates.

$$L = L \cup \{k\}$$

4. For all i not belonging to L , set $D_i = \min[D_i, D_k + D_{ki}]$
5. Goto step 3

Step 1 removes the low bandwidth path and steps 2-5 find a path satisfying delay constraint.

Chapter 3

Thesis Objective

In particular this thesis investigates 2 classes of QoS routing algorithms, namely resource reservation algorithm and dynamic algorithms. In resource reservation algorithms, packets are sent on a particular path from a source once the path is computed and reserved for the source to destination communication. And in dynamic algorithms, a path is chosen from a node to a neighbor node on an ad-hoc basis till the packet reaches the destination. Distance Vector Algorithm is a dynamic algorithm whereas BFM and Dijkstra's algorithms are resource reservation algorithms. We take a look at DVA and BFM as they represent an algorithm from each class. As BFM is faster and more efficient than Dijkstra's, we chose BFM in this study. Our objective is to propose a secure DVA and secure BFM algorithm.

The protocols are validated defining security and QoS metrics. The two main objectives of this thesis are:

1. Propose secure BFM and DVA QoS algorithms
2. To evaluate the effectiveness of these secure protocols.

Chapter 4

Design

4.1 Assumption:

We make certain assumptions for this study.

1. All the links within the network is one-directional
2. All the nodes are numbered
3. The initial bandwidth of each of the links is supplied manually
4. The system knows all the location of the filter locations

Two Routing Algorithms are proposed in the following subsections. The first algorithm is based on the BFM algorithm and the second algorithm is based on the Distance Vector algorithm.

4.2 Input:

The algorithms work on a direct network structures with filters. Each node is numbered. We assume that we know the bandwidth available on each of the links (denoted by labels on each link).

The network in the diagram below displays an example of a network structure. Here, the filters are displayed in bold color. This network diagram is used for the explanation of the protocols.

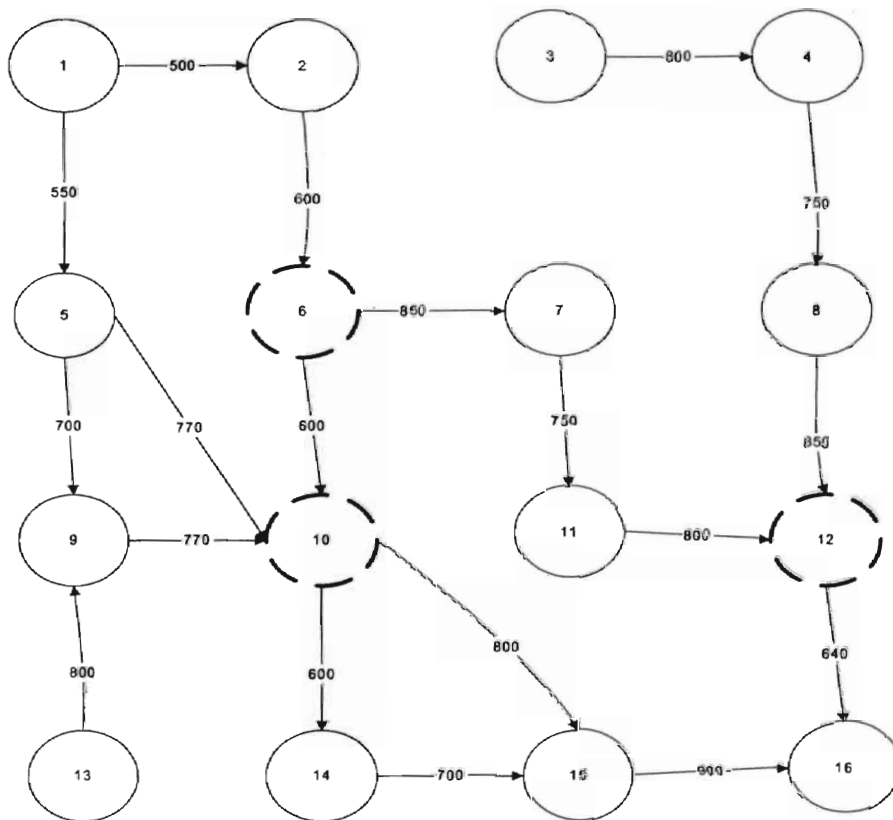


Figure 3: Input Example

4.3 Secure BFM Algorithm:

This algorithm finds a secure path from a source node to destination and reserves the path for future communication. It follows the Bellman-Ford-Moore algorithm to find most secure path. This protocol can be used for multimedia communications.

In this case, there are fewer chances for a denial of service attack. The attacker must reside within the reserved set of nodes. But it takes more time to find a path from source to destination, as we need to first find the path, then establish connections and then send packets. Once a path is established from source to destination, only that discovered path can be used to send packets from the particular source to the destination. To send packets on a new path, a new connection needs to be established.

4.3.1 A Metric for Security:

We need a security measure that can be used as a constraint along with bandwidth in this algorithm. We may find the security measure by going through an example.

In our input example:

We have the nodes 6, 10 and 12 as filters.

7 paths from node 1 to node 16

Path A 1-2-6-10-15-16

Path B 1-2-6-7-11-12-16

Path C 1-2-6-10-14-15-16

Path D 1-5-10-15-16

Path E 1-5-9-10-15-16

Path F 1-5-10-14-15-16

Path G 1-5-9-10-14-15-16

From each link, a filter can determine its coverage:

Filter 6 has only one input link – covers nodes 1 and 2

Filter 12 has only two input links – link 1 covers nodes 7 and 11, link 2 covers nodes 3, 4 and 8

Filter 10 has three input links – link 1 covers nodes node 6 (which is itself a filter), link 2 covers 1 and 5, and link 3 covers nodes 1, 5, 9, 13

Let $S(F_{i,l})$ be the set of nodes covered by filter i on link l .

Therefore,

$$S(F_{6,l}) = 1,2$$

$$S(F_{12,l}) = 7, 11$$

$$S(F_{10,1}) = \{ \} \text{ (empty)}$$

$$S(F_{10,2}) = 1, 5$$

$$S(F_{10,3}) = 1, 5, 9, 13$$

Given this scheme the spoofable nodes are:

Path A 1, 2, 15

Path B 1, 2, 7, 11

Path C 1, 2, 14, 15

Path D 1, 5, 15

Path E 1, 5, 9, 13, 15

Path F 1, 5, 14, 15

Path G 1, 5, 9, 13, 14, 15

Let P_A be the set of nodes in a path A

Let $\bigcup_{i=1}^n F_{i,A}$ be the set of n filters in a path A

Let D be the destination node

Let $\bigcup_{i=1}^n S(F_{i,A})$ be the spoofable set of nodes in a path

For example in path B ,

Path B 1-2-6-7-11-12-16

$$S(F_{6,1}) = 1, 2$$

$$S(F_{12,1}) = 7, 11$$

In other words,

$$S(F_{6,1}) \cup S(F_{12,1}) = \{1, 2, 7, 11\} = \bigcup_{i=1}^n S(F_{i,A}) \text{ for path B}$$

The security measure for a path is therefore:

$$P_A - (\bigcup_{i=1}^n F_{i,A} \cup D) \cup (\bigcup_{i=1}^n S(F_{i,i}))$$

The best security path is therefore:

$$\max(((P_A - (\bigcup_{i=1}^n F_{i,A} \cup D)) \cup (\bigcup_{i=1}^n S(F_{i,i}))) \dots\dots\dots (1)$$

4.3.2 Modified Secure BFM algorithm:

In the following discussion scanning a node v means examining all the edges (u,v) and labeling the neighbor nodes of u, if possible. A sweep refers to the process of scanning all the nodes as 1,2,3,...,n in that order.

Following is an implementation of the algorithm using the concept of sweep.

1. Initialization
2. Sweep
3. If for no node P_A value gets updated, STOP.

Initialization:

The source node (suppose node 2) is named A₁. A₁ names the nodes in its forwarding table as A₂, A₃ ... and so on. If there is no alias for the destination node, the algorithm fails.

Then we update the filter coverage accordingly.

For example when source is node 1:

Filter coverage for node 6 at link 1 is : {A₁, A₂}

Filter coverage for node 12 at link 1 is: {A₇, A₁₀}

Filter coverage for node 12 at link 2 is: {3, 4, 8}

Here, A₁ is an alias for node 1. Both can be used to denote node 1.

Let s be the source node. Initially P_A is $\{s\}$ for the source node s and $\{1,2,\dots,\infty\}$ for the rest of the nodes. A_1, A_2, \dots, A_n are the nodes in consideration in the initialization stage. So, for any node v except s , cardinality $|P_A(v)| = \infty$.

Sweep Phase:

Pick any link (u,v) such that $|(P_A(u) - (\bigcup_{i=1}^n F_{i,A(u)} \cup D) \cup (\bigcup_{i=1}^n S(F_{i,l})))| > |P_A(v)|$ and

(Bandwidth(v) \geq bandwidthRequired).

$P_A(v) = P_A + v$ (where v is not a filter and not the destination).

If no such link is available, the BFM algorithm terminates. At the termination P_A provides the s - t path.

We provide an example of this algorithm in the next section.

4.3.3 Traceback:

Once the path is established any node (except the filters and destination) in the path can spoof. In this way the number of nodes that can spoof decreases.

Equation: Suppose a is an attacking node.

$$a \in (P_A - (\bigcup_{i=1}^n F_A + D))$$

For $\bigcup_{i=1}^n F_A$ finds the number of filter for path P_A and D is the destination.

4.4 Secure Distance Vector Algorithm:

This algorithm finds a secure QoS path from a source to destination within a network. It follows the Distance Vector Algorithm to find the most secure path. The QoS metrics used are hops and bandwidth. Initially each node floods a hello packet throughout the network. A hello packet contains the node id, the number of hops,

number of regular nodes and the number of filters the packet has traversed till now. Any node receiving a hello packet updates the hop number, number of regular nodes and number of filters on the path; and forwards the packet to its incoming links. Though we assume that the user-defined network is uni-directional, the hello packets are allowed to traverse in the opposite direction. The following diagram displays the traversal path of hello packets initiated by node 16.

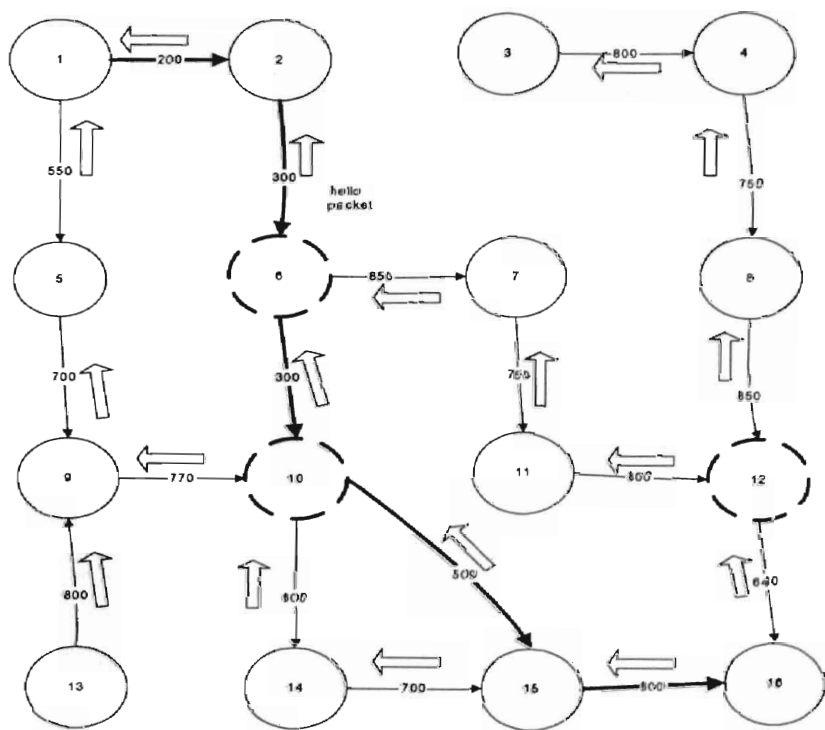


Figure 4: Hello Packet travel path

Upon receiving the hello packet, each node updates its forwarding table. A forwarding table contains information about a destination, next hop towards the destination and the total hops required to reach the destination, the number of regular nodes on the path and the number of filters on the path.

Next is a forwarding table of source node 1 containing necessary information to send packets towards a destination.

Destination	Next Hop	Hop	Regular nodes	No. of filters
2	2	1	1	0
3	{}	0	0	0
4	{}	0	0	0
5	5	1	1	0
6	2	2	2	0
7	2	3	2	1
8	{}	0	0	0
9	5	2	2	0
10	2	3	2	1
10	5	3	3	0
11	2	4	3	1
12	2	5	4	1
13	{}	0	0	0
14	2	4	2	2
14	5	4	3	1
15	2	4	2	2
15	5	5	4	1
16	2	5	3	2
16	5	5	4	1

Table 1: Forwarding Table for node 1

The rest of the nodes also create its own forwarding in similar fashion.

We may have multiple entries for a destination in the forwarding of a node. This only means that there are multiple paths from this node towards the destination. In that case, a node will forward the packet to a neighbor based on bandwidth available and hops required. Also, it sends the packet to a neighbor through which it takes the minimum number of regular nodes and the maximum number of filters to traverse to reach the destination. If two or more paths have similar metrics, then a node chooses one of the

paths on a random decision. It has been mentioned earlier that each node is assumed to know about each of its outgoing links.

Next we provide a flow diagram showing how a packet from a source node reaches the destination node.

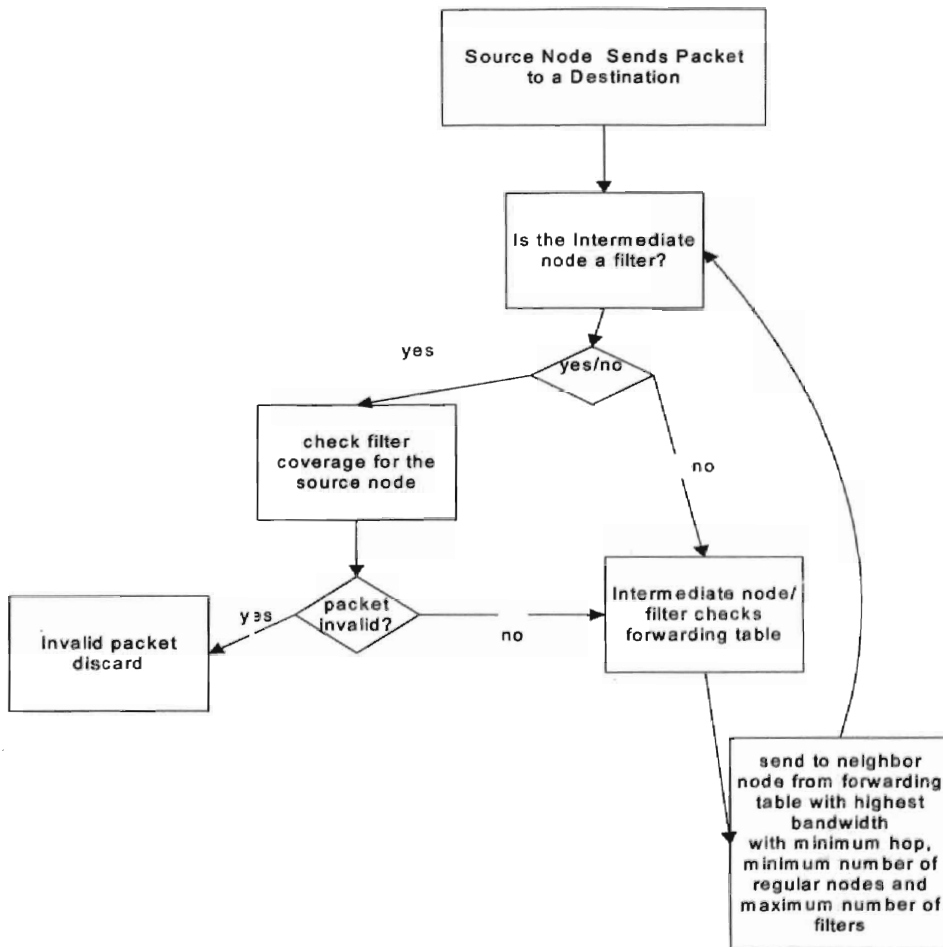


Figure 5: Flow Diagram for Secure DVA

The following section provides an example of this algorithm.

4.4.1 Denial of Service (DoS) Attack:

DoS attack can occur if one or more nodes start spoofing on behalf of another node and start sending packets to a destination. If the attacking nodes are already in the filter coverage, the filters would not block the inflow of packets towards the destination. As a result, the bandwidth will be consumed on the particular links. Therefore, the network communication is highly likely to be freezed on those particular links. Important messages sent by other nodes will be dropped and fail to reach the destination. If multiple node target a destination, then distributed denial of service attack will occur. Thus, the destination node may have to shut down its operation shortly because the incoming packets may overflow its queue.

4.4.2 Traceback:

The problem is that we cannot pinpoint any specific attacker in the case of denial of service attack. We just can make a prediction based on the filter coverage of the spoofed node.

However, we are certain that the attacker/s are members of the union of filter coverage for the spoofed node.

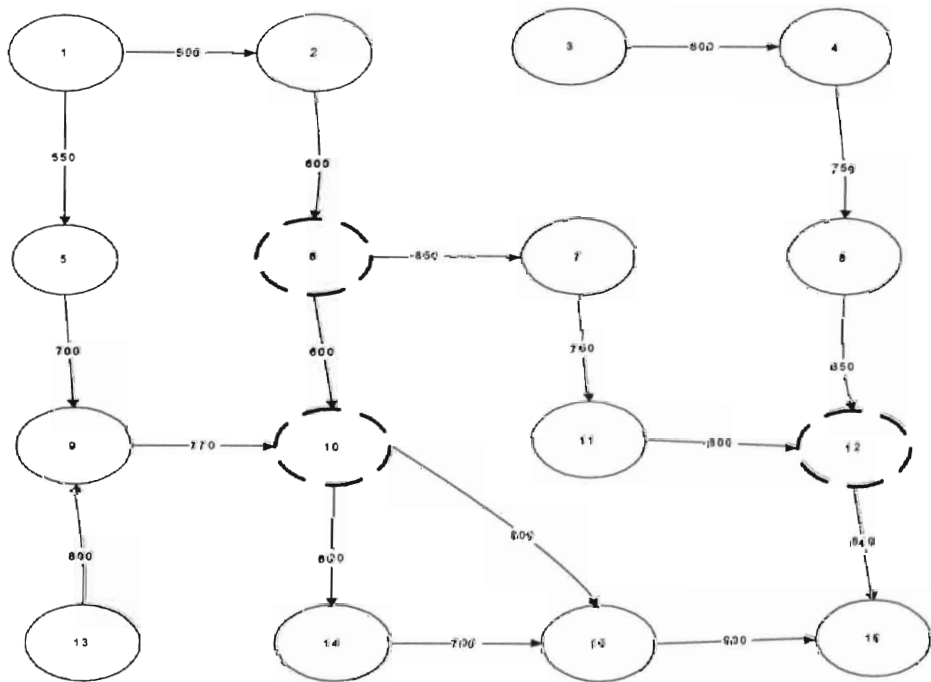
$$\text{Attacker, } a \in \bigcup S_k(F_{i,l})$$

Here, k is the source node. $S_k(F_{i,l})$ gets the filter coverage for filter i on link l for source node k .

Chapter 5

Implementation Approach

The protocols would be tested for several network systems. Visual C++.net would be used for the implementation purpose and the platform would be Windows 98/2000/XP. We assume that the user provides the network structure and the location of the filters. We consider that the links between nodes are uni-directional. An example of an input network is given below:



5.1 Protocol Design:

5.1.1 Finding out neighbors:

We assume that each node only keeps information about its first degree neighbors. Initially each of the nodes forwarding table is empty. Each node sends hello packets to its incoming links. We assume that hello packets can be sent in the opposite direction. Each node updates its neighbor table after receiving the “hello” packets.

A hello packet contains basic Information about the sender node. Once receiving a hello packet, a node easily learns about the source.

For example:

Node 1 receives hello packets from node 2 and node 5. Thus node 2 and node 5 notify node 1 about their presence. Therefore, forwarding table for node 1 is: {2,5}

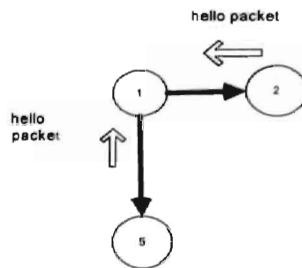


Figure 6: Neighbor Table Construction

5.1.2 Finding out Filter Coverage for a node:

At first the user selects the source node and the destination node. Initially the source node sends out control messages towards the filter. Each node receiving the control packet adds its node id to P_A and then forwards it to its neighbor nodes.

Let $S(F_{i,l})$ be the set of nodes covered by filter I on link l .

Initially $S(F_{i,l})$ is empty for all the filters.

Once a filter receives P_A , it updates its spoofable set for the source node. It takes non-filter nodes the nodes in P_A after the last filter specified in P_A . The filter sends control message containing P_A to all its outgoing links.

Once this is done, all the other nodes that were not included in P_A sends out control messages with their node id. $S(F_{i,l})$ is updated after each filter receives P_A from these nodes.

We assume that the filter coverage same for both protocols.

5.2 Secure Routing Protocol Based on Bellman-Ford-Moore Algorithm:

5.2.1 Packet Fields:

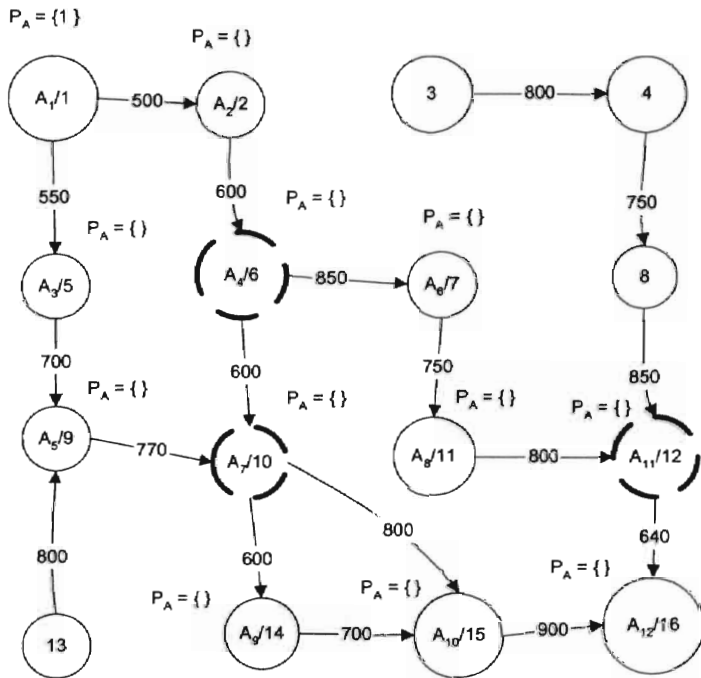
(Source id, P_A , Destination id, cid, bandwidth Requirement)

Here, cid is the job id. P_A is the set of all nodes on path A. Initially the user specifies the bandwidth requirement for the connection. A node receiving a request checks for P_A . If it is on the path, it discards the request as it finds a loop.

The following is an example of this protocol where the source node is 1 and destination node 16 and Bandwidth required = 300.

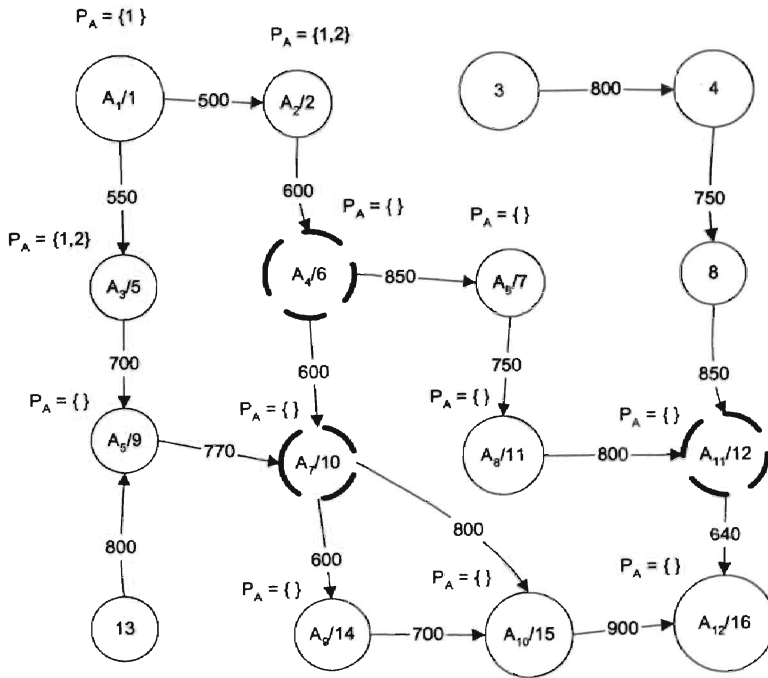
Initialization:

Bandwidth required = 300



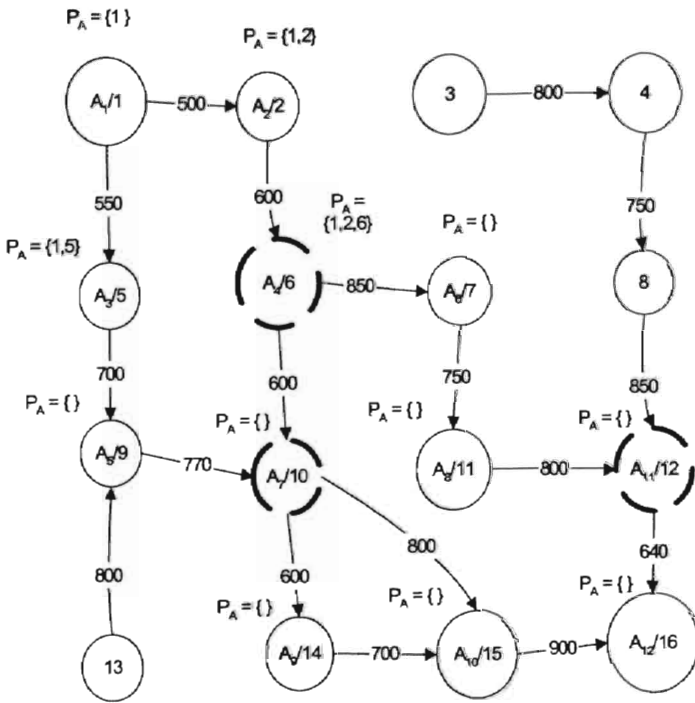
Initially Source node A_1 has $P_A = \{1\}$ and rest of the nodes has $P_A = \{1, 2, \dots, \infty\}$.

Sweep 1:



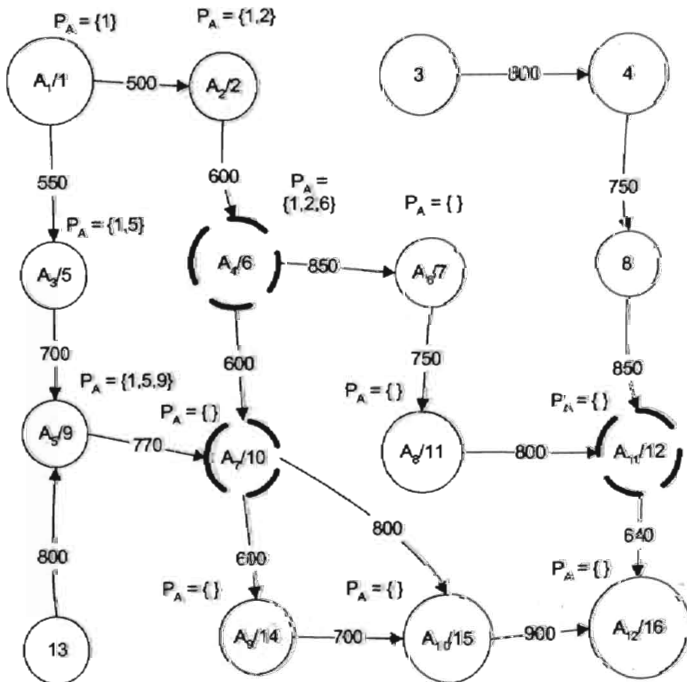
A_1 sends packet to A_2 and A_3 . $|P_A|$ at A_2 and A_3 is greater than $\{1\}$. So, A_2 updates P_A to $\{1,2\}$ and A_3 updates P_A to $\{1,5\}$.

Sweep 2:



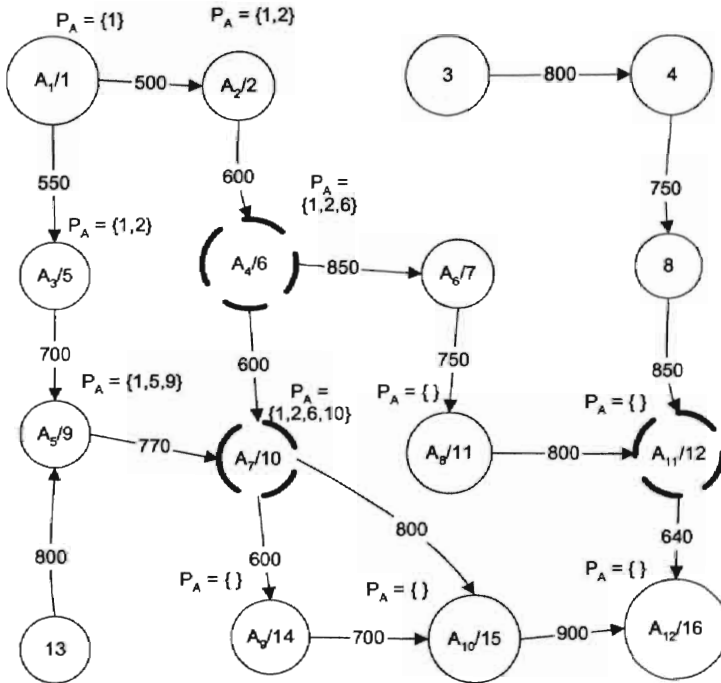
A_2 sends packet to A_6 (filter). As A_6 is a filter P_A remains unchanged at A_6 .

Sweep 3:



A_3 sends P_A to A_5 . $|P_A|$ at $A_3 = \infty$, as a result at A_3 P_A becomes $\{1,5,9\}$

Sweep 4:



A_6 sends P_A to A_7 (filter). P_A remains unchanged at A_7 .

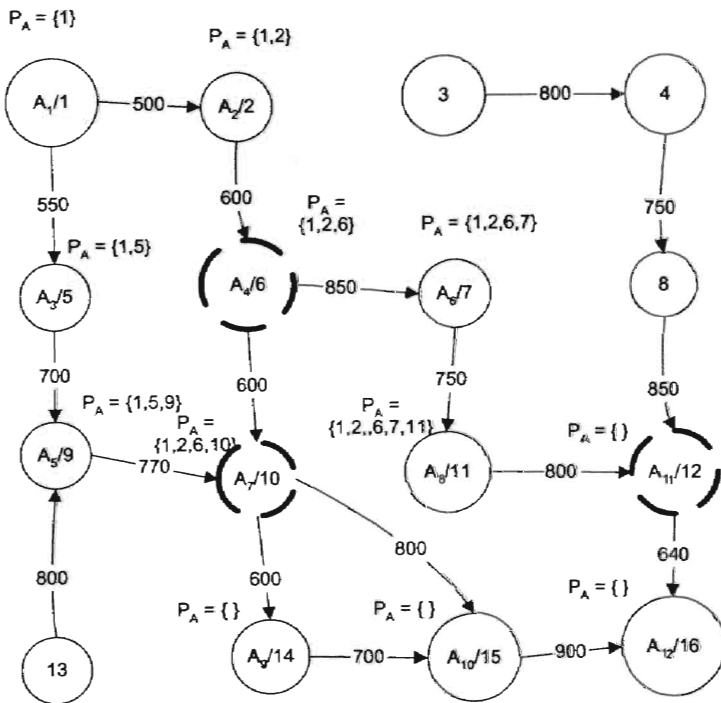
Sweep 5:

State unchanged

A_5 sends P_A to A_7 . $|P_A|$ at A_5 is 3, whereas $|P_A|$ at A_7 is 5. So P_A remains unchanged at

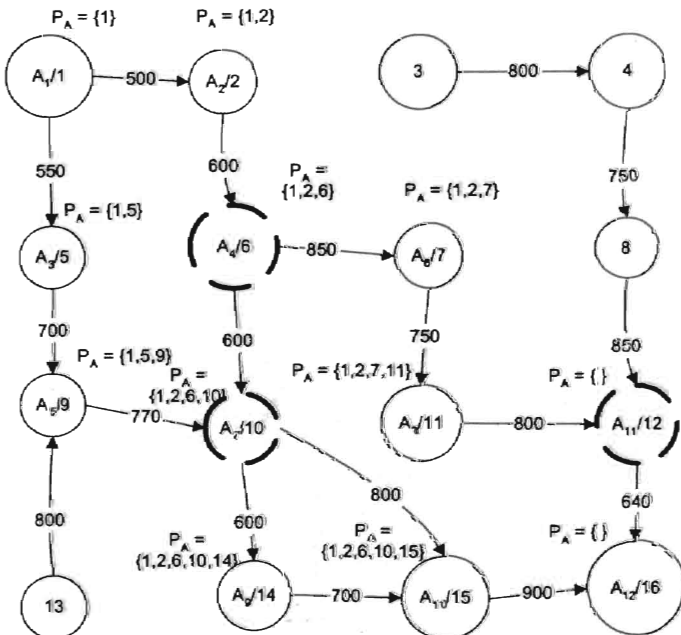
A_7 .

Sweep 6:



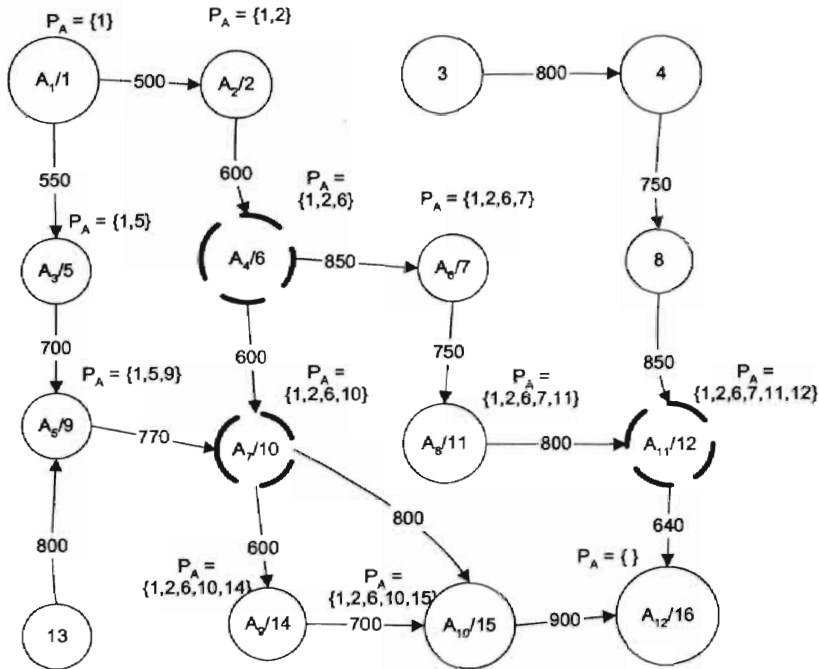
A_6 sends P_A to A_8 . P_A at A_8 get updated to $\{1,2,7,10\}$

Sweep 7:



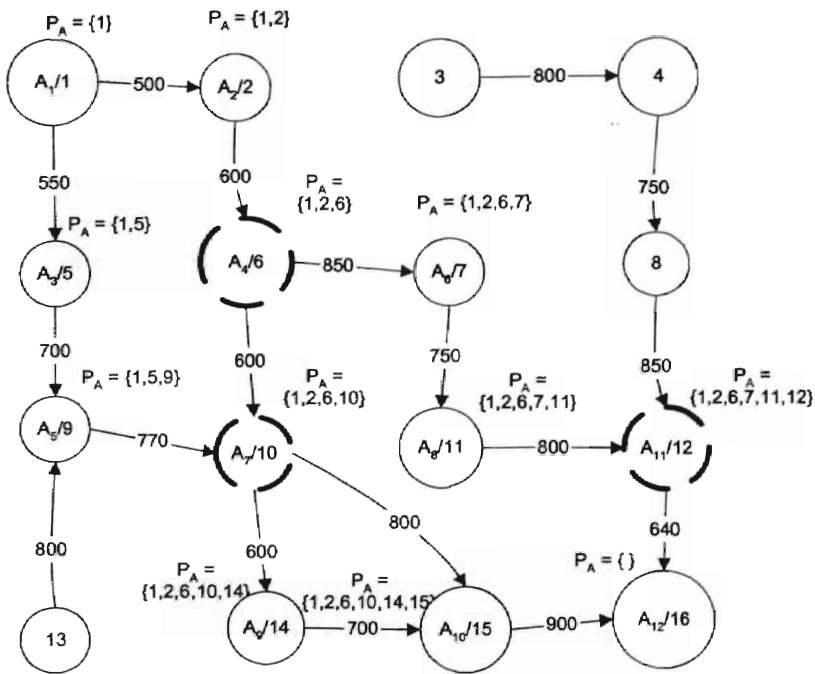
A_7 sends P_A to A_9 and A_{10} . P_A gets updated for both the nodes.

Sweep 8:



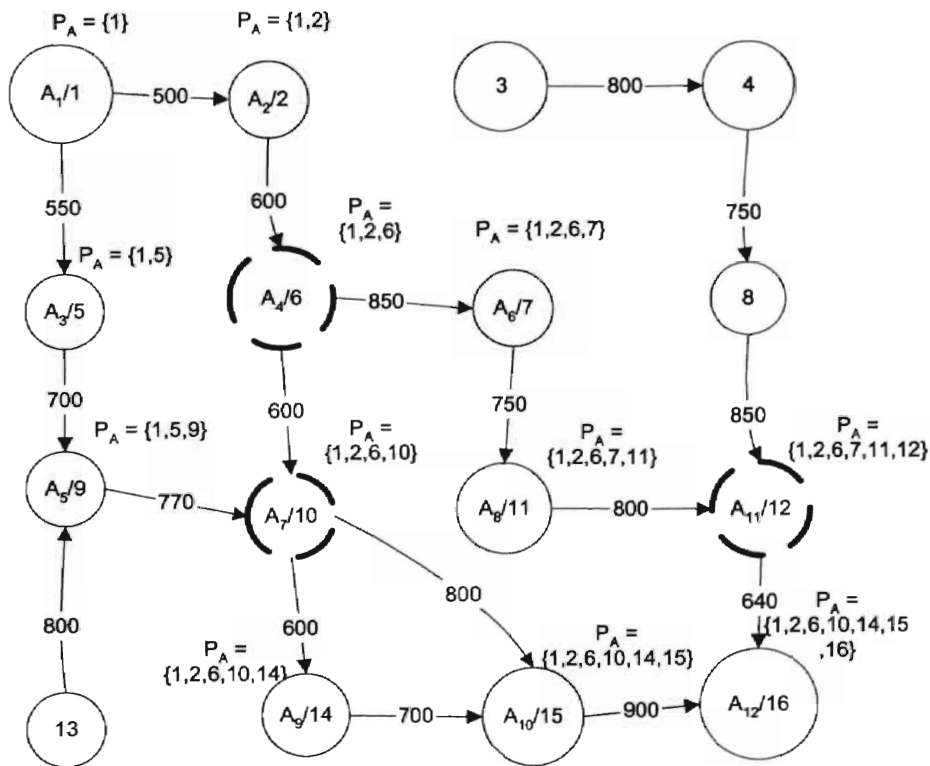
A_{11} is a filter. Upon Receiving P_A from A_8 , A_{11} updates it P_A to $\{1,2,6,7,10\}$

Sweep 9:



A_9 sends P_A to A_{10} . $|P_A|$ at A_{10} is less than the received P_A . P_A is changed at A_{10} .

Sweep 10:



A_{10} sends P_A to destination A_{12} . A_{12} updates its P_A to $\{1,2,6,10,14,15,16\}$.

Sweep 11:

State unchanged.

A_{11} sends P_A to A_{12} . But P_A sent by A_{11} can have 4 attacking nodes on its path whereas the current P_A at A_{12} can have 6 attacking nodes on its path. So, P_A remains unchanged at A_{12} .

Thus, the path from node 1 to 16 is $\{1, 2, 6, 10, 14, 15, 16\}$ which is the most secured path available according to secure BFM.

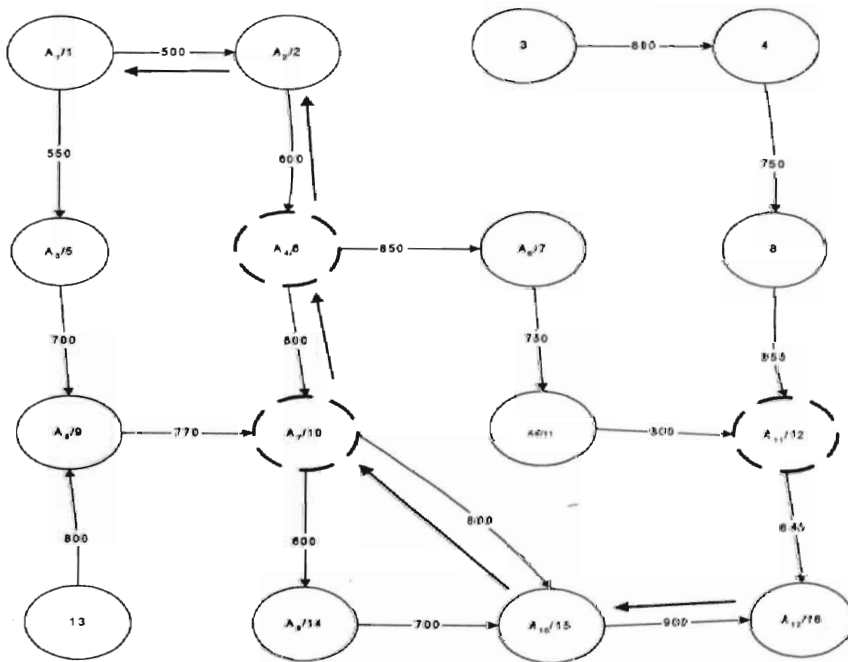
If necessary, a hop constraint can be added.

The destination sends back confirmation to reserve the path based on PA. While we reserve, each node on the path updates its bandwidth available. When done each node on the path releases the bandwidth.

5.2.2 Path reservation:

Reservation of the path P_A starts from the destination in the reverse order. For each link the bandwidth is reserved for the particular job. The reserved bandwidth is released once information transfer is terminated. The following diagrams describe the reservation procedure.

Step 1:



Step 2:

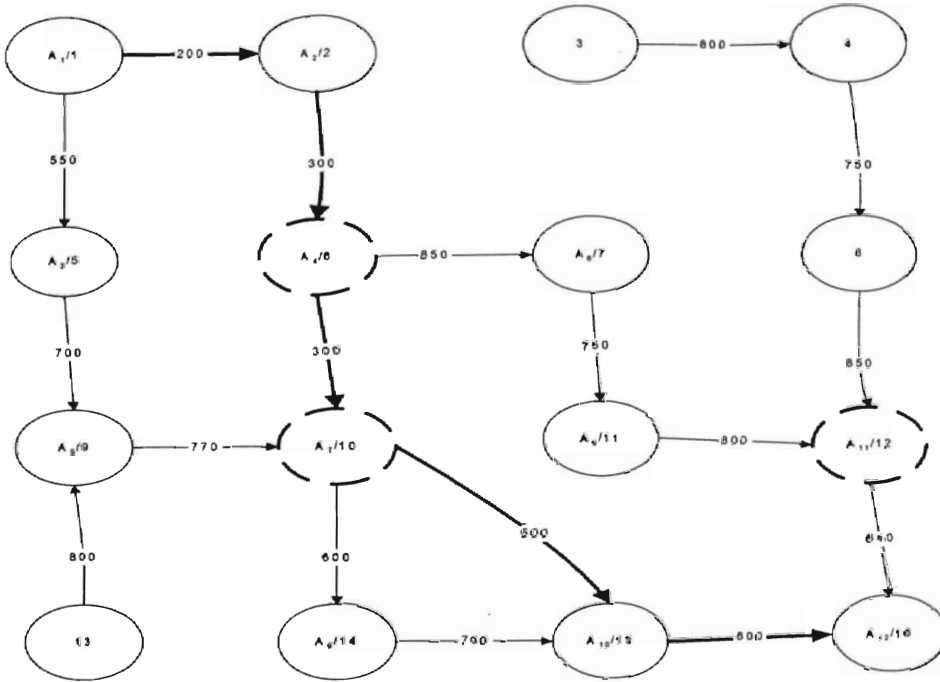


Figure 7: Reservation of a path

5.2.3 Traceback:

Once the path is established any node (except the filters and destination) in the path can spoof. In this way the number of nodes that can spoof decreases. Suppose: for the path 1-2-6-10-15-16 (as per example)

$$P_A = \{1,2,6,10,15,16\}, \bigcup_{i=1}^n F_A = \{6,10\} \text{ and } D = 16.$$

From the equation, $a \in (P_A - (\bigcup_{i=1}^n F_A + D))$, we find that 'a' can be a member of

the set $\{1,2,15\}$. That means, only nodes 1,2 and 15 can spoof on behalf of node 1 on this path.

5.3 Secure Routing Protocol Based on Distance Vector Algorithm:

5.3.1 Packet Field:

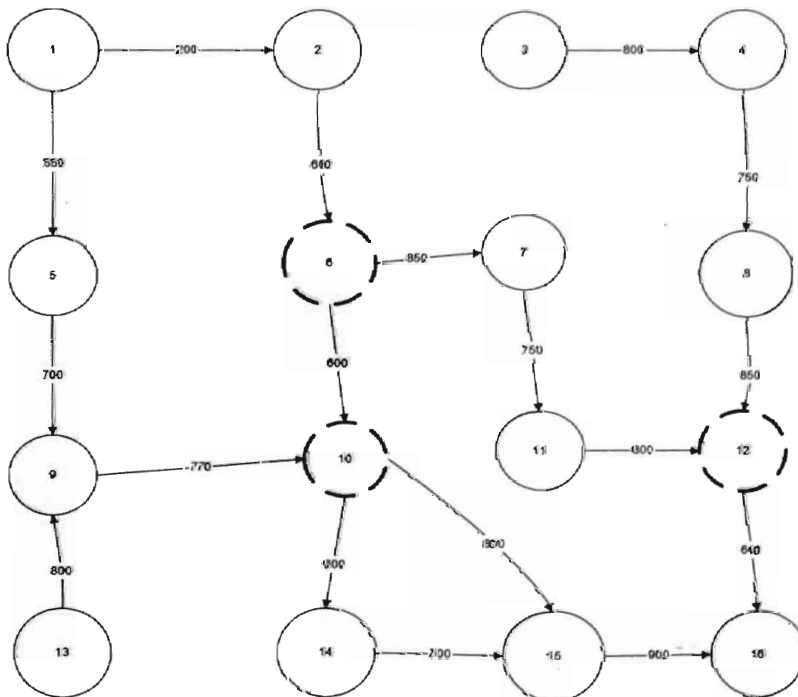
(source id, destination id, Bandwidth Requirement)

Here, bandwidth requirement refers to the bandwidth requirement for sending a packet on a link. As we can see, the bandwidth decreases with each packet on the network and bandwidth is released once the packet reaches the neighboring node.

Filter coverage is the same as the proposed algorithm based on the BFM algorithm covered earlier in this paper.

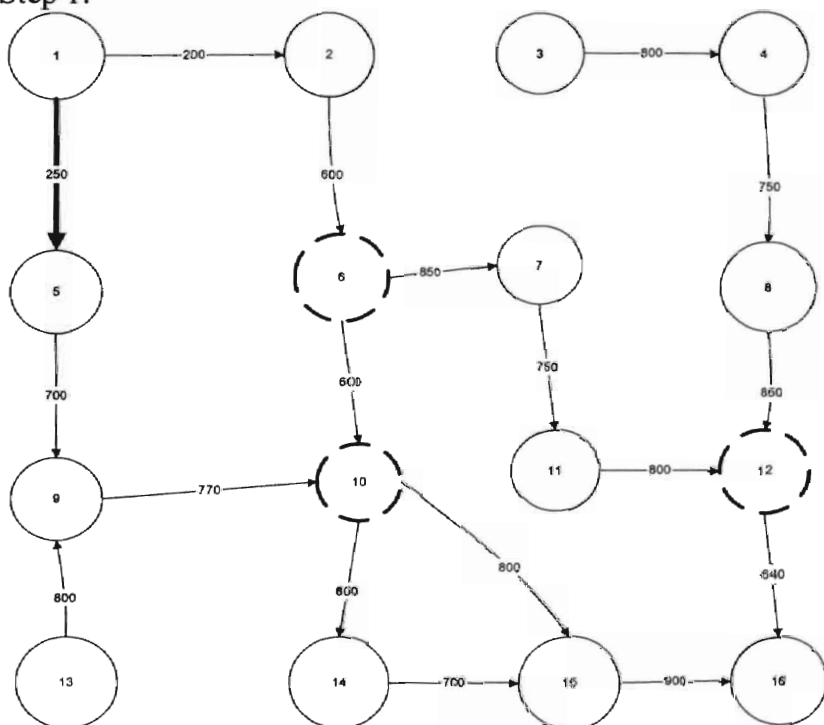
The following is an example to find a path from source node 1 to destination node 16. We use the proposed protocol based on Distance Vector Algorithm to find the path. The network being used is user-defined.

User-defined Network: Bandwidth required = 300.



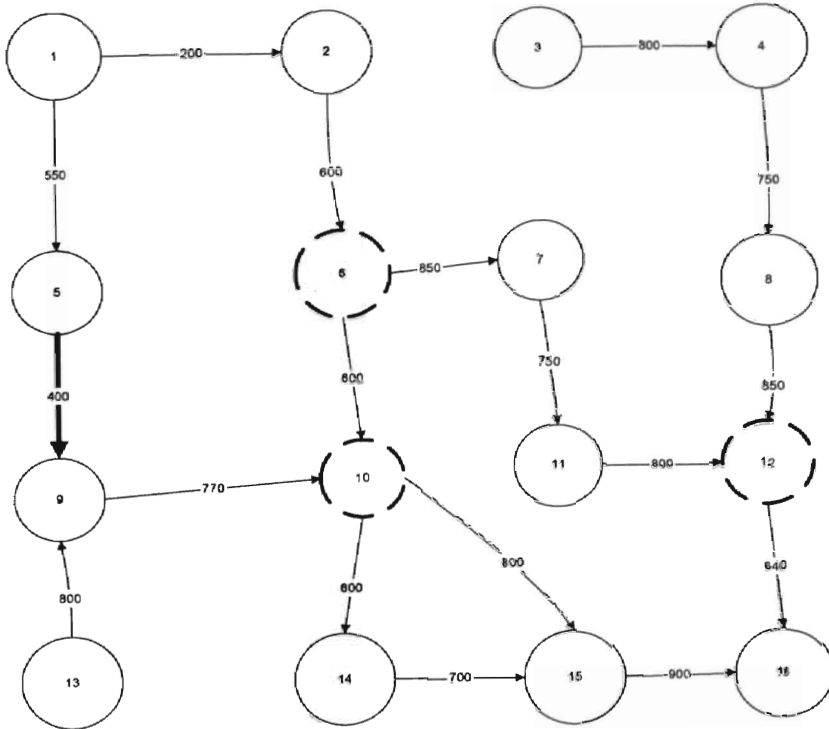
Initial network with numbered nodes and labeled bandwidth on each link.

Step 1:



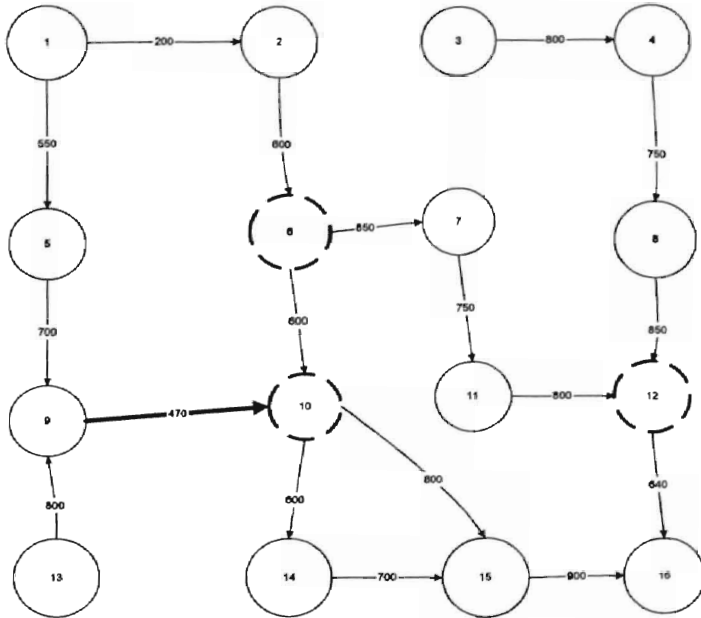
Bandwidth on (1,2) is 200 which is less than 300. So, the Packet is sent on the path from node 1 to 5. Bandwidth is reduced from 550 to 250.

Step 2:



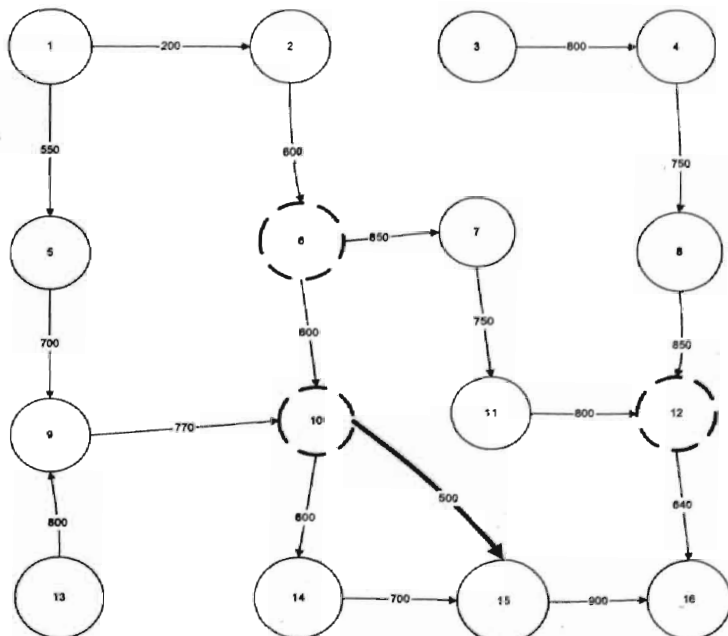
Packet reaches node 5. Bandwidth is returned to link (1,5).

Step 3:



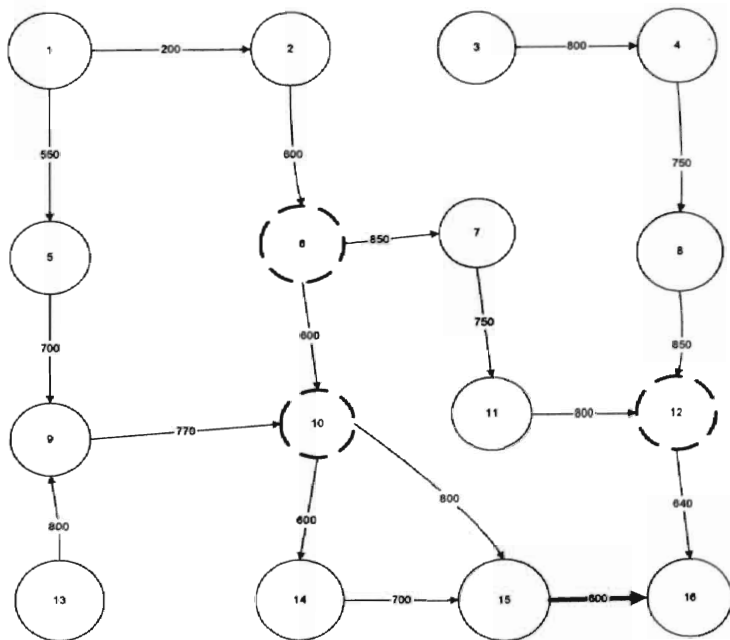
From node 9 the forwarding node to go to destination 16 is node 10 which is a filter. Bandwidth is reduced to 470. Filter 10 checks the validity of the packet. It receives the packet on its link 1, which is a valid path for source node 1. Bandwidth is returned to the link (9,10).

Step 4:



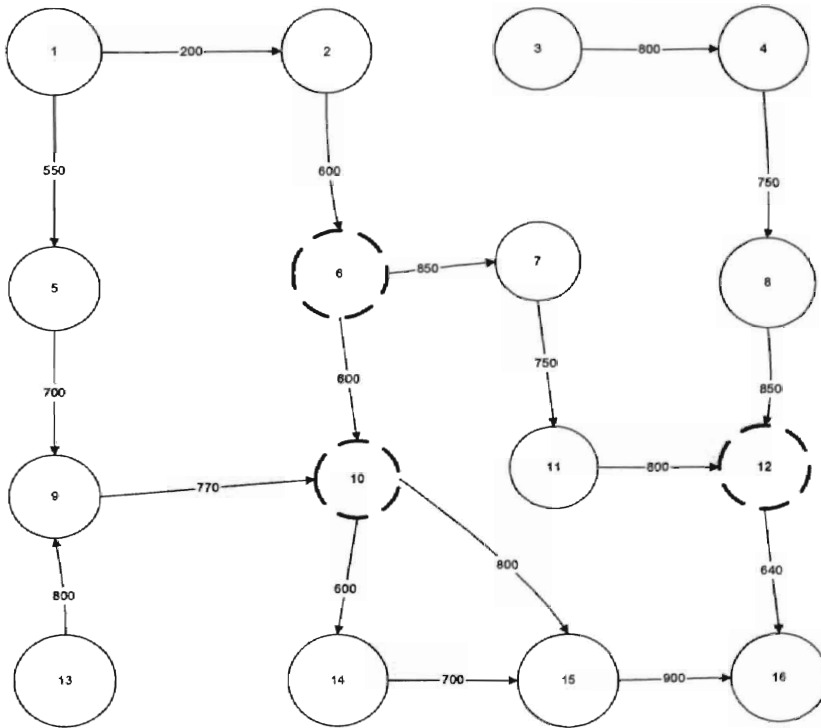
From node 10 the forwarding node is node 15 to reach destination. The link (10,15) has enough bandwidth to support the packet transfer. So, the packet is sent to node 15. Bandwidth is reduced to 500 while packet is on the link.

Step 5:



Bandwidth is returned to the link (10,15). Node 15 forwards packet to the destination.

Step 6:



Packet reaches destination node 16.

In Step 1, node 1 sends packet to node 5 instead of node 2. The reason behind is that the link between node 1 and 2 has bandwidth less than 300. It is clear that node 2 would have been chosen if there were enough bandwidth available between node 1 and node 2.

Chapter 6

Simulation Results and Analysis

Network topologies were generated using the BRITE Topology Generation Tool [17]. A hundred simulations were performed in each category and the average taken.

Networks with 20 nodes, 40 nodes, 60 nodes, and 80 nodes were generated using BRITE.

To generate a network, Waxman's probability model [18] for interconnecting the nodes of the topology was used. The model is given by:

$$P(u, v) = \alpha e^{-d/(BL)}$$

Where $0 < \alpha, \beta \leq 1$, d is the Euclidean distance from node u to node v , and L is the maximum distance between any two nodes.

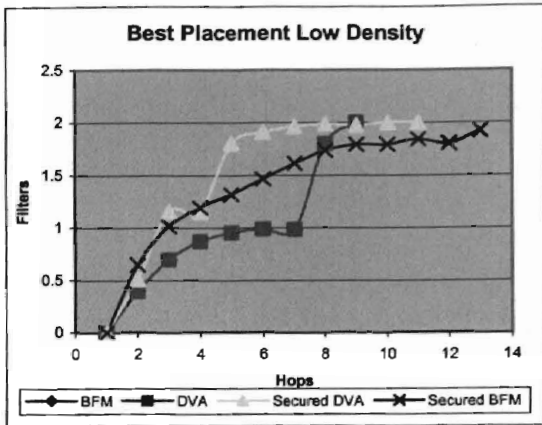
For each network, 4 kinds of simulations were done.

1. Best Placement Low Density
2. Best Placement High Density
3. Random Placement Low Density
4. Random Placement High Density

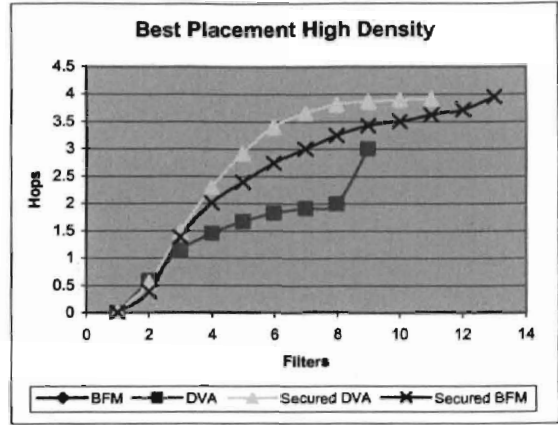
At low density, only 10% of all nodes are filters. 20% of all filters are considered filters at high density. In the case of best filter placement strategy, the filters were placed strategically at nodes with the maximum input and output links. In random filter placement strategy, the filters were placed in a random manner.

For each edge, bandwidth capacity was generated with a random distribution between 10 and 1024.

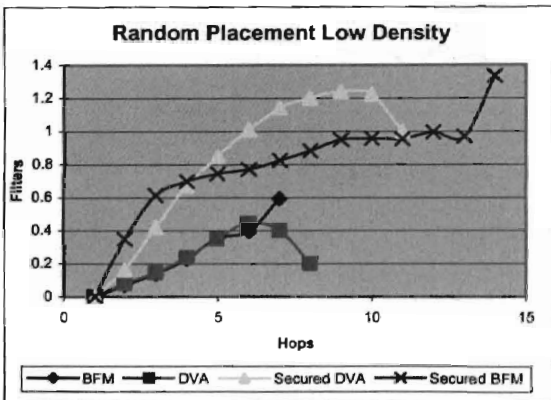
First we have a look at Figure 8(a) for 20-node network in the case of best filter placement low density. 2 (10% of 20) nodes were assigned as filters because of low density filter placement. The secure DVA performs the best among all the algorithms. It



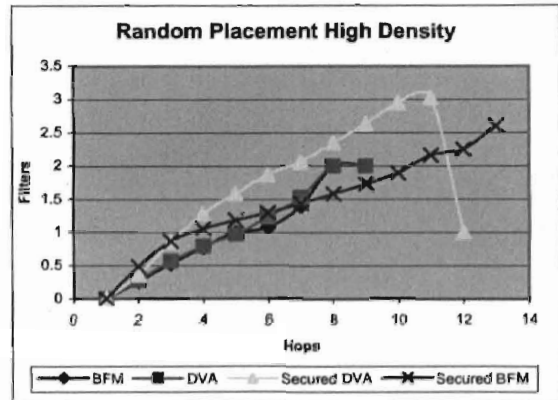
(a)



(b)



(c)



(d)

Figure 8: Security on 20-node network

finds more filters on paths of shorter lengths. There seems to be only three graphs because BFM and DVA generated the same results. We can see that secure BFM performs fairly well. It generates longer paths than secure DVA and also there is a graduate increase in filter quantity in terms of path length. Equation 1 (page 23) doesn't find the path with maximum filters on some occasions. The Equation depends on the

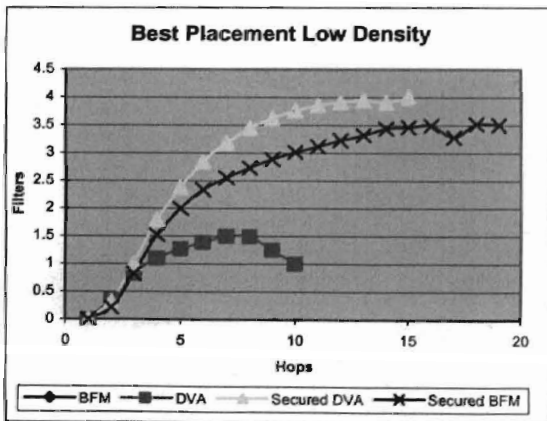
filter coverages. And we may get higher filter coverage for lesser number of filters due to the randomness of the network. That's why, secure BFM doesn't reach the 2 filter mark. We can see hikes on secure DVA, BFM and DVA because of randomly selected paths.

Next, we look at Figure 8(b) for a 20-node network with best filter placement at high density. 4 (20% of 20) nodes were assigned as filters because of high density filter placement. The secure DVA performs best among all the algorithms. The secure BFM finds longer paths but performs in a consistent manner. BFM and DVA find the shortest paths and also finds less filters on various paths compared to secure DVA and secure BFM. Secure DVA finds more filters on shorter paths. Whereas the performance of secure BFM depends on the random behavior of Equation 1 (page 23). That's why it finds close to maximum filters on some occasions.

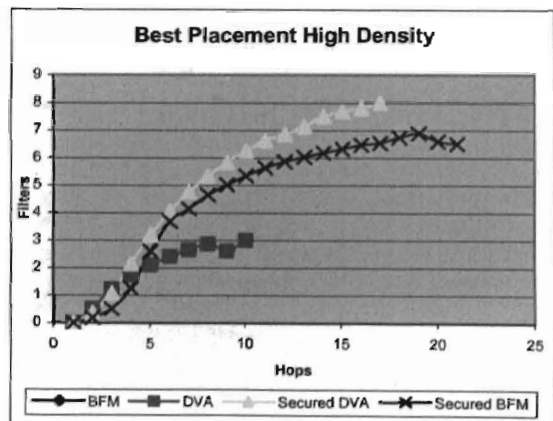
Now, let's consider Figure 8(c) for a 20-node network with random filter placement at low density. 2 (10% of 20) nodes were randomly assigned as filters because of low density filter placement. The graphs indicate that the number of filters in the path increases with the path length upto a certain point. Below this optimum point, the number of filter in a path decreases even as the path length increases. This is particularly true of the secure DVA algorithm. Secure DVA performs quite well but falters at the end. The sharp drops in secure DVA and DVA are seen because of randomly selected paths. However, future work is required to confirm this. Moreover the secure BFM performs well for longer paths. This happens due to the random behavior of Equation 1 (page 23). Equation 1 indicates that paths of longer hops with more filters in them will be generated for secure BFM.

Now, we consider Figure 8(d) for a 20-node network with random filter placement at high density. 4 (20% of 20) nodes were randomly assigned as filters because of high density filter placement. We see a sharp drop at the end for the secure DVA. We see this sharp drop because of randomly selected paths and also the filters were placed randomly. Further research is needed to confirm this. Secure BFM does fairly well for longer paths. The BFM and DVA perform in a consistent manner. They find the shortest path between two nodes. And the filters found on each path depend on the randomness of this filter placement strategy. These results indicate that for long paths, secure BFM is the preferred choice, whereas for short and medium paths, secure DVA provides better security.

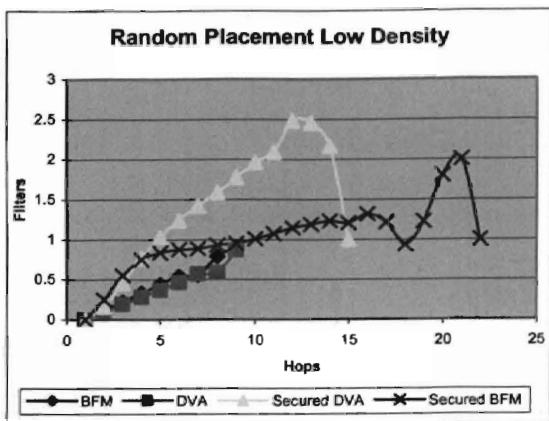
Next, we consider Figure 9(a) for a 40-node network with best filter placement at low density. The secure DVA performs best among all the algorithms. This



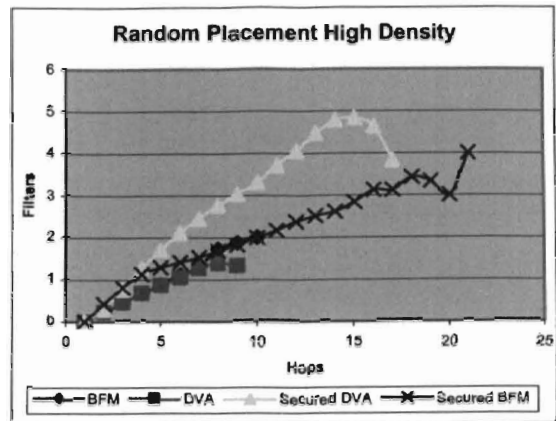
(a)



(b)



(c)



(d)

Figure 9: Security on 40-node network

happens because secure DVA tries to reduce the number of regular nodes and at the same time increase the number of filters. As a result, it produces shorter paths with more filters compared to secure BFM. Secure BFM also performs pretty well. Because of Equation 1 (page 23) it produces larger hops for secure BFM but on some occasions it does not find the maximum number of filters on a path. Again, as expected, the DVA and BFM find shortest paths amongst the algorithms. We see a sudden drop in the number of filters for these algorithms at hop 8. This happens because the paths were selected randomly.

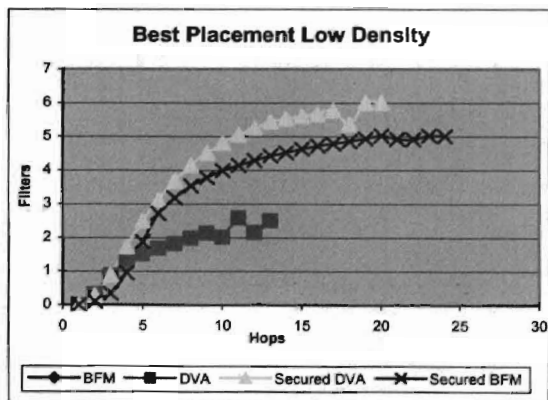
Figure 9(b) is considered for a 40-node network in the case of best filter placement at high density. The performance of the algorithms is very similar to Figure 9(a). Because of high density filter placement, 8 (20% of 40) nodes were assigned as filters. For longer hops it is noticeable that Secure DVA and Secure DVA find more filters on a path compared to DVA and BFM.

Next, we look at Figure 9(c) for a 40-node network with random filter placement at low density. 4 (10% of 40) nodes were randomly assigned as filters. Secure DVA starts off doing quite well. But we see a sharp drop after hop 12 because of randomly

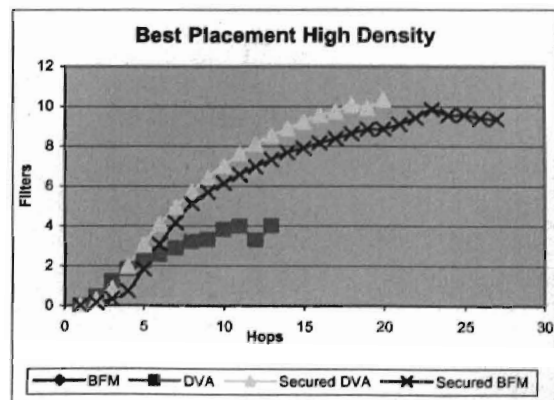
positioned filters within the network. This effect has been noticed with smaller and larger networks and is an area for future research. Secure BFM does better than the DVA and BFM. We also see a few drops and hikes on secure BFM because of randomly selected paths. BFM and DVA find the shortest paths with fewer filters among the algorithms.

Now, let's look at Figure 9(d) for a 40-node network in the case of random filter placement at high density. 8 (20% of 40) nodes were randomly assigned as filters. Secure DVA performs the best among all the algorithms. Secure BFM produces longer hops compared to secure DVA. Due to Equation 1 (page 23) it does not find the maximum filters on its paths. The sharp drops and hikes on secure DVA and secure BFM are due to the random selection of paths. We see slight difference in BFM and DVA because of the filters was selected randomly for each of the algorithms.

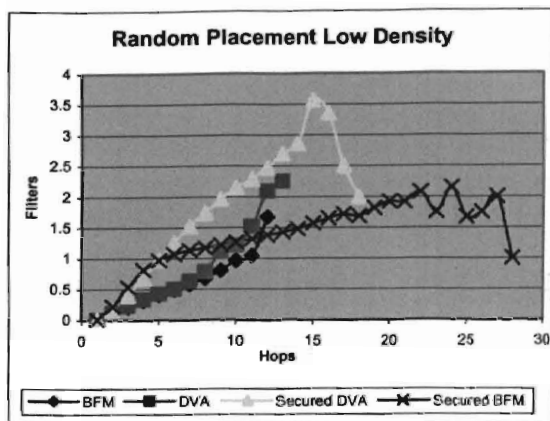
Next, we look at Figure 10(a) for a 60-node network with best filter placement at low density. In this figure secure DVA performs the best among all the algorithms. Secure DVA generates shorter paths with more filters on each path. On the



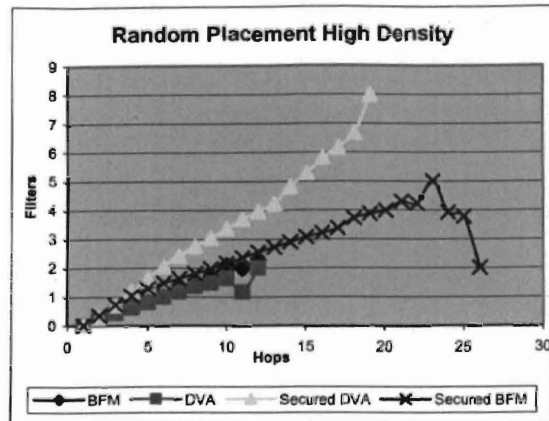
(a)



(b)



(c)



(d)

Figure 10: Security on 60-node network

other hand, secure BFM produces longer paths with fairly good amount of filters on each path. It happens because of Equation 1 (page 23). The DVA and BFM finds shortest path with fewer filters on each path. The hikes and drops in secure DVA, DVA and BFM happen due to the randomness of the paths.

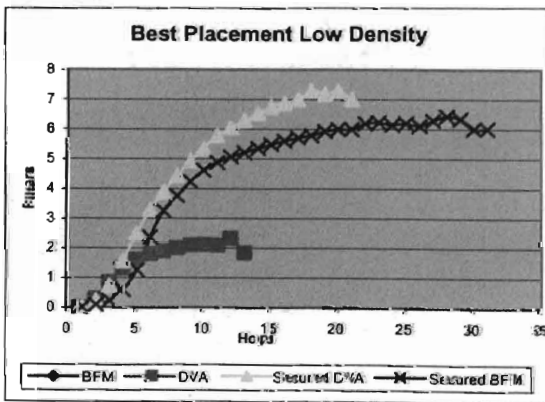
Figure 10(b) is shown for a 60-node network in the case of best filter placement at high density. 12 (20% of 60) nodes were assigned as filters. We see almost the same picture as Figure 10(a) with more filters on each path because of high density filter placement.

Now, we look at Figure 10(c) for a 60-node network with random filter placement at low density. 6 (10% of 60) nodes were randomly placed as filters. Secure DVA starts performing quite well but has a sharp drop at the end. This happens due to randomly placed filters. Secure BFM performs worst. We see some hikes and drops in the case of secure BFM. This happens because of randomly selected paths and randomly positioned filters. The DVA and BFM falls in between secure DVA and secure BFM at the end. DVA and BFM find the shortest paths.

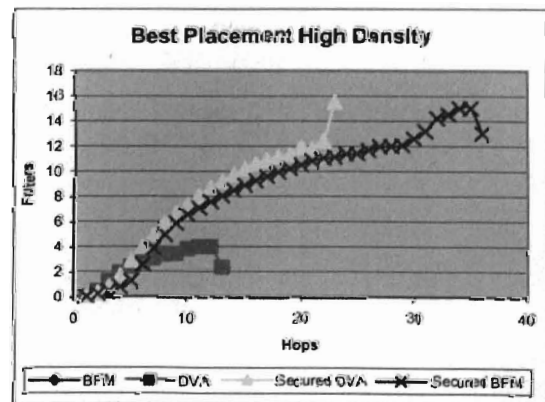
Next, we consider Figure 10(d) for a 60-node network in the case of random filter placement at high density. 12 (10% of 60) nodes were randomly assigned as filters. Secure DVA performed the best among all the algorithms. Secure BFM performed gradually but its performance dropped at the end. It happened due to randomness of Equation 1 (page 23). The DVA and BFM found the shortest paths between any two nodes but with fewer filters on each path.

Figure 11(a) shows the graph for an 80-node network with best filter placement at low density. 8 (10% of 80) nodes were assigned as filters. The performance of secure DVA is shown to be the best. Secure DVA reduces the number of regular nodes and also tries to increase the number of filters on a particular path. Therefore, it generates shorter paths with more filters. Secure BFM also performed well. It found longer paths because of Equation 1 (page 23). It found a good quantity of filters on its paths. The DVA and BFM found the shortest paths with fewer filters on the paths.

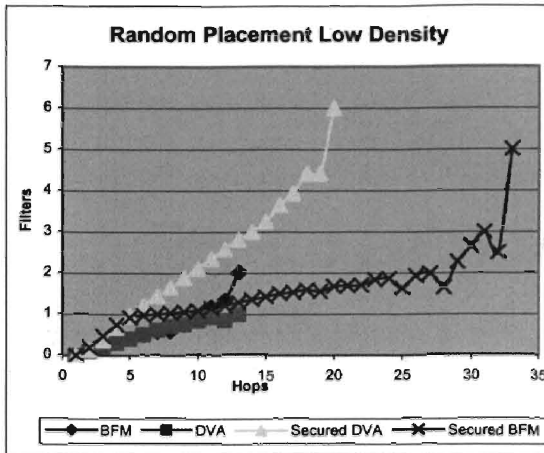
Now we look at Figure 11(b) for an 80-node network in the case of best filter placement at high density. 16 (20% of 80) nodes were assigned as filters in this case.



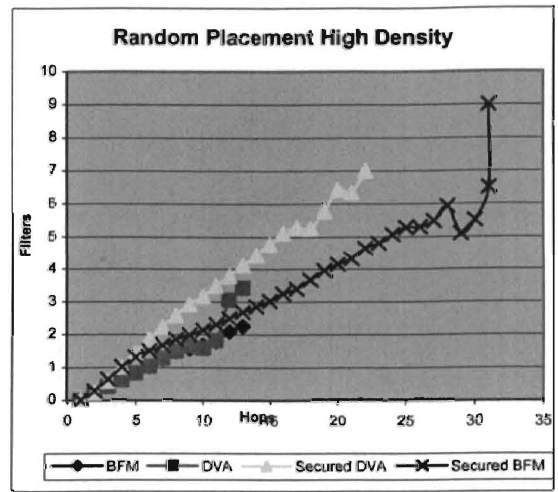
(a)



(b)



(c)



(d)

Figure 11: Security on 80-node network

The performance of DVA was still the best. It found the shorter paths with more filters on the paths. Secure BFM found longer paths with a good amount of filters between two nodes. We see a drop for secure BFM because of the randomness in Equation 1 (page 23). The DVA and BFM found the shortest paths. These paths had fewer filters in them.

Lets look at Figure 11(c) for an 80-node network in the case of random filter placement at low density. 8 (10% of 80) nodes were randomly assigned as filters. Secure DVA performs best among all the algorithms. Secure BFM doesn't do well till hop 28. After that we see some hikes on secure BFM. This happens due to the randomness of Equation 1 (page 23). The BFM and DVA find the shortest paths among all the algorithms. We see a hike of BFM due to randomly selected paths.

Now, we look at Figure 11(d) for an 80-node network with random filter placement at high density. 16 (20% of 80) nodes were randomly assigned as filters. Secure DVA again performed well. We see a better performance of secure BFM compared to Figure 11(c). This happens because there are more filters within the

network now. Equation 1 (page 23) produces larger paths. And on larger paths we see a better performance in terms of filters in the case of Secure BFM. The DVA and BFM find the shortest paths. We see a hike on DVA due to randomly selected paths.

The following main conclusion can be drawn from these simulations:

- As expected, the shortest routes are obtained using the standard BFM and DVA
- Again as expected the best placement strategy produced more secure routes than a random placement strategy
- Simulations show the superiority of the secure DVA over the secure BFM algorithm.
- An unexpected result is the degradation in performance of the secure DVA as the number of nodes in the network exceeds a certain threshold. In other words, there is an optimum path length at which the number of filters in the path is a maximum. Above this length, the number of filters in the path decreases. But as the number of nodes in a network increase, secure DVA tends to overcome this behavior. Further research is needed to define the 'optimum' point when the number of filters in the path is a maximum.

Chapter 7

Conclusions

In this paper we have proposed secure QoS routing algorithms for dynamic routing and resource reservation. The paths generated by these algorithms are longer than those generated by non-secure algorithms. The secure Distance Vector Algorithm provides shorter paths and better security than the secure Bellman-Ford-Moore algorithm. That is, the path generated by the secure DVA passes through more filters, thus reducing the scope for spoofing. We get more secure paths using best filter placement with high density. Further work is needed to determine the optimal distribution of filters in the network. Future work will also investigate secure routing satisfying multiple QoS requirements.

This work may be extended to investigate IP traceback to determine the probable locations of the attackers. A real system is more desired to carry on the tests. Also, experiments should involve as many nodes as possible.

However, in summary we believe that the secure QoS routing algorithms can significantly reduce the Denial of Service attacks.

REFERENCES

- [1] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, 1990
- [2] Eric Greenberg, *Network Application Frameworks: Design and Architecture*, Addison Wesley, 1999
- [3] Donna Ghosh, Venkatesh Sarangan and Raj Acharya, *Quality-of-Service Routing in IP Networks*, IEEE Transactions on Multimedia, VOL. 3, NO. 2, June 2001, Pages :200-208
- [4] Kihong Park and Heejo Lee, *On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets*, Proceedings of the 2001 Conference on Applications, Technologies, Architecture and Protocols for Computer Communication, ACM SIGCOMM, page(s) 15-26
- [5] Jaikumar Vijayan, *Denial of Service Attacks on the Rise?* URL: <http://www.cnn.com/2002/TECH/internet/04/09/dos.threat.idg/> (April 9, 2002)
- [6] S. Gibson, *The Strange Tale of the Denial of Service Attack against GRC.com*, URL: <http://grc.com/dos/grcdos.htm> (March 5, 2002)
- [7] Mark Ward, *Web Warning Center in Net Attack*, URL: <http://news.bbc.co.uk/1/low/sci/tech/1348820.stm> (May 24, 2001)
- [8] John T. Moy, *OSPF Anatomy of an Internet Routing Protocol*, Addison Wesley, 1998
- [9] *Overview of Scans and DDoS Attacks*, URL: <http://www.nipc.gov/ddos.pdf>
- [10] TechTarget, *WhatI?com*, URL: www.whatis.com

- [11] L. Stein. The World Wide Web Security FAQ, version 3.1.2, URL:
<http://www.w3.org/Security/FAQ/> (February 4, 2002)
- [12] Jupitermedia Corporation, *Online Dictionary For Computer and Internet terms*,
URL: www.webopedia.com (October 13, 2002)
- [13] R.K.Ahuja, T.Magnanti and J.B.Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, 1993
- [14] Danielle Dunne, *What is a Denial of Service Attack?*, URL:
<http://www.darwinmag.com/learn/curve/column.html?ArticleID=115&action=print>,
(June 24, 2001)
- [15] Brent Baccala, *Distance Vector Algorithms, An Internet Encyclopedia*, URL:
www.freesoft.org/CIE/RFC/1058/6.htm (April 1997)
- [16] Jon Crowcroft, Zhang Wang, *Quality of Service Routing for Supporting Multimedia Applications*, IEEE Journal on Selected Areas in Communication, 14(7), pages 1288-1234, Sep 1996
- [17] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers, *BRITE: An Approach to Universal Topology Generation*, Proceedings of IEEE MASCOTS '01, August 2001
- [18] B. Waxman, *Routing on Multipoint Connections*, IEEE Journal on Selected Areas on Selected Areas in Communication, December 1988

APPENDIX A

GLOSSARY

Node: In a network, a node is a connection point, either a redistribution point or an end point for data transmissions. In general, a node has programmed or engineered capability to recognize and process or forward transmissions to other nodes.

Spoof: To deceive for the purpose of gaining access to someone else's resources (for example, to fake an Internet address so that one looks like a certain kind of Internet user)

Bandwidth: Bandwidth (the width of a band of electromagnetic frequencies) is used to mean (1) how fast data flows on a given transmission path, and (2), somewhat more technically, the width of the range of frequencies that an electronic signal occupies on a given transmission medium..

Hop: One hop is defined as the transit through one router. Each router always adds 1 to account for itself.

ICMP: Short for Internet Control Message Protocol, an extension to the Internet Protocol (IP). ICMP supports packets containing error, control, and informational messages. The PING command, for example, uses ICMP to test an Internet connection.

Ping: A utility that determines whether a specific IP address is accessible. It works by sending a packet to the specified address and waiting for a reply. PING is used primarily to troubleshoot Internet connections.

APPENDIX B

IMPLEMENTATION

```
//Munirul Islam
//OSU ID# 440-15-5382
//Thesis

/*****/

// project.cpp

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <windows.h>

#include "nodes.h"
#include "statistics.h"

const char STRATEGIES[][80]=
{
    "Best placement, low density",
    "Best placement, high density",
    "Random placement, low density",
    "Random placement, high density",
    "Incremental placement, low density",
    "Incremental placement, high density"
};

const char ALGORITHMS[][80]=
{
    "Print all paths",
    "Bellman-Ford-Moore Algorithm",
    "Distance Vector Algorithm",
    "Secure Distance Vector Algorithm",
    "Secure Bellman Ford Moore Algorithm"
};

struct setup
{
    int placement_type;
    int path_strategy;
    float bandwidth;
};
```

```

void eatFile(const char* fileName, ofstream& report,
read_apply_search(cssetup& setup, mask, ofstream& report,
Statistics& stat, setup& setup);
int main(int argc, char**argv)
{
    if(argc!=3)
    {
        cout <<"Syntax: "<<argv[0]<<" path\mask*.brite
report_file\n\n";
        return 1;
    }
    ofstream report(argv[2]);
    if(!report)
    {
        cout <<"Can't open report file "<<argv[2]<<endl;
        return 1;
    }

    setup setup;

    /* placement stuff */
    cout << "\nSpecify strategy for placement of the
filters:\n";
    cout << "1. Best placement, low density\n";
    cout << "2. Best placement, high density\n";
    cout << "3. Random placement, low density\n";
    cout << "4. Random placement, high density\n";
    cout << "5. Incremental placement, low density\n";
    cout << "6. Incremental placement, high density\n";

    /* read placement type (repeat until user made correct
selection) */
    int placement_type;
    cin >>placement_type;
    while(placement_type<1 || placement_type>6)
    {
        cout <<"Value of "<<placement_type<<" is incorrect choice.
Retype: ";
        cin >>placement_type;
    }
    setup.placement_type = placement_type;
    report <<"Placement type: "<< STRATEGIES[placement_type-
1]<<endl;

    /* read path selection strategy*/
    cout << "\nSpecify path selection strategy:\n";
    cout << "b) Bellman-Ford-Moore Algorithm\n";
    cout << "c) Distance Vector Algorithm\n";

```

```

cout << "e) secure Bellman Ford Moore Algorith \n";
cout << "d) secure Distance Vector Algorithm\n";
char *p;

do
{
    cin >> menu_selection;
    while(cin.get() != '\n') ;

    p=strchr("BCDE", toupper(menu_selection));
    if(p==NULL)
    {
        cout <<"Wrong selection. Retype: ";
    }
}
while(p==NULL);

static const char menumap[]={1,2,2,3,4};
report <<"Algorithm: " << ALGORITHMS[toupper(menu_selection)-
'A'] <<endl;
setup.path_strategy=menumap[toupper(menu_selection)-'A'];

cout <<"Ender minimum bandwidth: ";
cin >> setup.bandwidth;
report <<"minimum bandwidth: " << setup.bandwidth <<endl;
report <<"files selected: " << argv[1] <<endl;

report <<endl <<endl;
report
<<"*****\n\n"
;
Statistics stat;
applySearch(argv[1], report, stat, setup);
return 0;
}

void applySearch(const char* mask, ofstream& report,
Statistics& stat, setup& setup)
{
    WIN32_FIND_DATA FileData; // Data structure describes the
file found
    HANDLE hSearch; // Search handle returned by
FindFirstFile
    BOOL bFinished = FALSE;

    hSearch = FindFirstFile( mask, &FileData);

```

```

{
    cerr <<"No files "<<mask<<" found\n";
    if (hSearch == INVALID_HANDLE_VALUE)
        return;
}

char *fileDirEnd=strrchr(mask, '\\');
const char *fileDir;
if(fileDirEnd)
{
    *(++fileDirEnd)=0;
    fileDir=mask;
}
else
    fileDir="";

while (!bFinished)
{
    char *fullFileName=new
char[strlen(fileDir)+strlen(FileData.cFileName)+1];
    strcpy(fullFileName, fileDir);
    strcat(fullFileName, FileData.cFileName);
    eatFile(fullFileName, report, stat, setup);
    delete[] fullFileName;

    if (!FindNextFile (hSearch, &FileData))
    {
        bFinished = TRUE;

        if (GetLastError () != ERROR_NO_MORE_FILES)
        {
            cerr <<"Unable to find next file within "<<mask<<endl;
        }
    }
}

// Close the search handle.

if (!FindClose (hSearch))
{
    cerr << "Unable to close search handle.\n";
}

stat.calculateGlobalStat();

report<<"\n\n\nRESULTS\n-----\n\n";
stat.reportGlobalStat(report);
report.flush();
}

```

```

void eatFile(const char* fileName, ofstream& report,
Statistics& stat, setup& setup)
{
    Nodes nodes;

    if(!nodes.loadNodes(fileName))
    {
        report<<fileName<<" : cannot be loaded (check file
syntax)\n";
        return ;
    }

    cout <<"Analyzing " <<fileName<<"..."<<endl;
    nodes.placeFilters(setup.placement_type);

    int start, end;

    stat.startNetwork();
    for(start=0; start<nodes.node_number; start++)
    {
        if(start) cout <<"\b\b\b\b";
        cout <<setw(3)<<100*start/(nodes.node_number-1)<<"%";
        cout.flush();
        if(nodes.nodes[start].isFilter()) continue;
        for(end=0; end<nodes.node_number; end++)
        {
            if(start==end || nodes.nodes[end].isFilter()) continue;

            //guaranteed start!=end, and both are non-filters

            Path path=nodes.findPath(start, end,
setup.path_strategy, setup.bandwidth,NULL);
            if(path.exists())
            {
                stat.addPath(path);
            }
        }
    }
    cout<<endl;
    stat.finishNetwork();
    report<<fileName<<endl;
    stat.reportNetwork(report);
    report.flush();
}

/*****

//node.h

```

```

#ifndef NODE_H
#define NODE_H

class node {

private:
    int node_id;
    double x,y;
    bool filter;
    int in_neighbors[50];
    int in_neighbor_count;
    int out_neighbors[50];
    int out_neighbor_count;
    double bandwidth[50];
    double distance[50];
public:
    node(void); // Default constructor.
    node(int,double,double); // Constructor
    ~node(void); // Destructor
    void setFilter(void);
    void clearFilter() {filter=false;}
    bool isFilter(void) const {return filter;}
    int get_in_neighbor_count(void) const;
    int get_out_neighbor_count(void) const;
    void set_in_neighbor(int);
    void set_out_neighbor(int,double, double);
    void operator=(const node&);

friend class Paths;
friend class Path;
};

#endif

/*****/

// node.cpp

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

//-----
// node methods
//-----

node::node(void) {
    node_id = -100;
    x = 0;
    y = 0;
}

```



```

int i;
for (i=0;i<50;i++)
    filter_in_neighbors[i] = -100;
for (i=0;i<50;i++)
    out_neighbors[i] = -100;
for (i=0;i<50;i++)
    {
        bandwidth[i] = -100;
        distance[i] = -100;
    }
in_neighbor_count = 0;
out_neighbor_count = 0;
}

node::node(int id,double x_coordinate,double y_coordinate) {
    node_id = id;
    x = x_coordinate;
    y = y_coordinate;
    filter = false;
    int i;
    for (i=0;i<50;i++)
        in_neighbors[i] = -100;
    for (i=0;i<50;i++)
        out_neighbors[i] = -100;
    for (i=0;i<50;i++)
        {
            bandwidth[i] = -100;
            distance[i] = -100;
        }
    in_neighbor_count = 0;
    out_neighbor_count = 0;
}

node::~node(void) {
}

void node::setFilter(void) {
    filter = true;
}

int node::get_in_neighbor_count(void) const {
    return in_neighbor_count;
}

int node::get_out_neighbor_count(void) const {
    return out_neighbor_count;
}

void node::operator=(const node& p)

```

```

        node_id = p.node_id;
        x = p.x;
        y = p.y;
        filter = false;
    int i;
        for (i=0;i<50;i++)
            in_neighbors[i] = -100;
        for (i=0;i<50;i++)
            out_neighbors[i] = -100;
        for (i=0;i<50;i++)
            {
                bandwidth[i] = -100;
                distance[i] = -100;
            }
        in_neighbor_count = 0;
        out_neighbor_count = 0;
    }

void node::set_in_neighbor(int i)
{
    in_neighbors[in_neighbor_count] = i;
    in_neighbor_count++;
}

void node::set_out_neighbor(int i, double bwidth, double dist)
{
    out_neighbors[out_neighbor_count] = i;
    bandwidth[out_neighbor_count] = bwidth;
    distance[out_neighbor_count] = dist;
    out_neighbor_count++;
}

/*****

// nodes.h

#ifndef NODES_H
#define NODES_H

#include "node.h"
#include "paths.h"

struct Nodes
{
    node *nodes;
    int *node_pos;
    int node_number;

    Nodes(): node_number(0), nodes(NULL), node_pos(NULL) {};

```

```

    bool loadNodes(const char *fileName);
    int placeFilters(int strategy);
    bool loadFilters(const {return node_number>0;}
    void showCovers(int start, void(* callback)(int, bool*));
    Path findPath(int start, int final, int algorithm, float
bandwidth, void(* callback)(Path&));
};
#endif

/*****

// nodes.cpp

#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "nodes.h"

bool Nodes::loadNodes(const char *fileName)
{
    char line[100];

    if(nodes) delete[] nodes;
    nodes=NULL;

    if(node_pos) delete[] node_pos;
    node_pos=NULL;

    ifstream in(fileName);
    if (!in)
    {
        return false;
    }

    node_number = 0;
    int i = 0, edge_number = 0, nodeNumber = 0;
    double x_axis = 0, y_axis = 0;
    in.getline(line,100);
    char *temp = strtok(line, " \t\n(),");
    // read number of nodes
    temp = strtok(NULL, " \t\n(),");
    node_number = atoi(temp);
    nodes=new node[node_number];
    temp = strtok(NULL, " \t\n(),");

    temp = strtok(NULL, " \t\n(),");
    edge_number = atoi(temp);
    in.getline(line,100);

```

```

in.getline(line,100);
in.getline(line,100);
matrix=new double*[node_number];

int **adj,**path;
adj = new int*[node_number];
path = new int*[node_number];

for (i=0;i<node_number;i++)
{
    matrix[i]=new double[node_number];
    adj[i] = new int[node_number];
    path[i] = new int[node_number];
    for (int j=0;j<node_number;j++)
    {
        matrix[i][j] = 0;
        adj[i][j] = 0;
        path[i][j] = 0;
    }
}
// network topology info processing
// while (!in.eof())
{
    for (i=0;i<node_number;i++)
    {
        in.getline(line,100);
        char* temp = strtok(line, " \t\n");
        char* num = new char[10];
        strcpy(num,temp);
        nodeNumber = atoi(num);
        temp = strtok(NULL, " \t\n");
        strcpy(num,temp);
        x_axis = double (atof(num));
        temp = strtok(NULL, " \t\n");
        strcpy(num,temp);
        y_axis = double (atof(num));
        long oldFlag = cout.setf(ios::fixed,ios::floatfield);
        long oldPrec = cout.precision(2);
        //cout << nodeNumber << " " << x_axis << " " <<
y_axis << "\n";
        cout.setf(oldFlag, ios::floatfield);
        cout.precision(oldPrec);
        node aNode(nodeNumber,x_axis,y_axis);
        nodes[i] = aNode;
    }
    cout.flush();
    do
    {

```

```

}
while(strcmp(line,"")==0);
for(i=0;i<edge_number;i++)
{
    in.getline(line,100);
    char* temp = strtok(line," \t\n");
    int from = 0, to = 0;
    double bw = 0, dist;
    temp = strtok(NULL," \t\n");
    from = atoi(temp);
    temp = strtok(NULL," \t\n");
    to = atoi(temp);
    nodes[to].set_in_neighbor(from);
    temp = strtok(NULL," \t\n");
    dist=(double) atof(temp);
    temp = strtok(NULL," \t\n");
    temp = strtok(NULL," \t\n");
    bw = (double) atof(temp);
    matrix[from][to] = bw;
    adj[from][to] = 1;
    nodes[from].set_out_neighbor(to,bw, dist);
    //cout << from << " " << to << " " << bw << "\n";
}
}

// filter placement
node_pos = new int[node_number];
int *neighborCount = new int[node_number];
for (i=0;i<node_number;i++)
    node_pos[i] = i;
for (i=0;i<node_number;i++)
    neighborCount[i] = nodes[i].get_in_neighbor_count() +
nodes[i].get_out_neighbor_count();

// bubble sort
for (i=0; i<node_number-1; i++) {
    for (int j=0; j<node_number-1-i; j++)
        if (neighborCount[j] < neighborCount[j+1]) { /*
compare the two neighbors */
            int tmp = neighborCount[j];    /* swap a[j] and
a[j+1] */
            neighborCount[j] = neighborCount[j+1];
            neighborCount[j+1] = tmp;
            tmp = node_pos[j];
            node_pos[j] = node_pos[j+1];
            node_pos[j+1] = tmp;
        }
}
for (i=0;i<node_number;i++)

```

```

        delete[] matrix[i];
    }
    delete[] matrix;
    delete[] adj;
    delete[] path;
    return true;
}

int Nodes::placeFilters(int placement_type)
{
    int
placement_nodes=(int)(node_number*(placement_type%2==1?0.1:0.2
));
    int period=node_number/placement_nodes;
    int position=period-1;
    int i;

    for(i=0; i<node_number; i++)
    {
        nodes[i].clearFilter();
    }

    /* randomize pseudo random numbers generator */
    srand(time(NULL));
    /* perform placement strategy */

    int really_placed=0;

    for(i=0; i<placement_nodes; i++)
    {
        if(placement_type<=2) /*if best placement*/
        {
            /* just put filter to the nodes with most neighbors
count */
            nodes[node_pos[i]].setFilter();
            really_placed++;
        }
        else if(placement_type<=4) /* random */
        {
            int position;
            // don't put filter flag twice: repeat trying until we
found a non-filter
            do
            {
                position=rand()%node_number;
            }
            while(nodes[position].isFilter());
        }
    }
}

```

```

        nodes[position].setFilter();
        really_placed++;
    } //place filter to the position found
    else if(placement_type<=6) /* incremental*/
    {
        if(position != node_number-1) /* not last node*/
        {
            nodes[position].setFilter();
            position+=period;
            really_placed++;
        }
    }
}
return really_placed;
}

```

```

Path Nodes::findPath(int start, int final, int algorithm,
float bandwidth, void(* callback)(Path&))

```

```

{
    Paths paths(nodes, node_number, 0);
    /* set path strategy for Paths object */
    paths.setStrategy(algorithm);

    /* set bandwidth for Paths object */
    paths.setBandwidth(bandwidth);

    /* run path selection algorithm*/
    if(algorithm>1)
    {
        Path bestPath=paths.find_paths(start, final);
        if(callback) callback(bestPath);
        return bestPath;
    }
    else
    {
        paths.setCallback(callback);
        return paths.find_paths(start, final);
    }
}

```

```

int Nodes::numFilters()
{
    int result=0;
    int i;
    for(i=0; i<node_number; i++)
    {
        if(nodes[i].isFilter()) result++;
    }
}

```

```

}

return result;
void Nodes::ShowCovers(int start, void(* callback)(int,
bool*))
{
    int i, j;
    bool *nodeCovered=new bool[node_number];

    for(i=0; i<node_number; i++)
    if(nodes[i].isFilter())
    {
        for(j=0; j<node_number; j++) nodeCovered[j]=false;
        nodeCovered[start]=true;
        Path path(nodes, node_number);
        path.markFilterCover(i, nodeCovered);
        nodeCovered[start]=false;
        callback(i, nodeCovered);
    }
    delete[] nodeCovered;
}

/*****

// paths.h

#ifndef PATH_H
#define PATH_H

#include "node.h"

//path contains directed path
class Path
{
    node* nodes;
    int count;

    int* path;
    int path_length;
    double path_distance;
    int count_filters;
    int securityMeasure;

    double edge_distance(int v, int w);
    void recalcSecurityMeasure();
public:
    Path();
    Path(node* nodes, int count);
    Path(const Path&);

```



```

bool is_in_path(int vertex);
void Path::add_vertex(int vertex);
void delete_from_end();
void print();

double getDistance() const {return path_distance;}
int getFiltersCount() const {return count_filters;}
int getLength() const {return path_length;}
bool exists() {return path_length>=2;}
Path& operator=(const Path& rhs);

int getItem(int index) {return path[index];}
int getSecurityMeasure();

void markFilterCover(int vertex, bool *nodeMeasured);
};

class Paths
{
    node* nodes;
    int* visited_from;
    int count;
    bool assume_unidirectional;

    int strategy;
    double bandwidth;

    Path minimumPath;
    void(* cb)(Path&);

    void find_path_between(int start, int finish, Path&
partial_path );

public:
    Paths(node* nodes, int count, bool assume_unidirectional);
    ~Paths();

    void setBandwidth(double bwidth) {bandwidth=bwidth;};
    void setStrategy(int strat) {strategy=strat;};
    void setCallback(void(* callback)(Path&)) {cb=callback;};
    Path find_paths(int start, int finish);
};

#endif

/*****/

// paths.cpp

```

```

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

//-----
// paths methods
//-----

Path::Path(node* nodes, int count):
    nodes(nodes), count(count), path_length(0),
    path_distance(0.0), count_filters(0),
    securityMeasure(-1)
{
    path=new int[count];
};

Path::Path():
    nodes(NULL), count(0), path_length(0), path_distance(0.0),
    path(NULL), count_filters(0),
    securityMeasure(-1)
{
};

/* copy constructor: required when re use path in return from
the function*/
Path::Path(const Path& rhs): path(NULL)
{
    *this=rhs;
}

Path::~Path()
{
    delete[] path;
}

/* copy operator: copies path from rhs to this */
Path& Path::operator=(const Path& rhs)
{
    if(this==&rhs) return *this;

    /* make sure to free current path if it's not NULL*/
    if(path) delete[] path;
    count=rhs.count;
    nodes=rhs.nodes;
    path_length=rhs.path_length;
    path_distance=rhs.path_distance;
    count_filters=rhs.count_filters;
    securityMeasure=rhs.securityMeasure;
}

```

```

/* replace path*/
if(rhs.path)
{
    path=new int[count];
    for(int i=0; i<path_length; i++) path[i]=rhs.path[i];
}
else
    path=NULL;

return *this;
}

double Path::edge_distance(int v, int w)
{
    for(int i=0; i<nodes[v].out_neighbor_count; i++)
    {
        if(nodes[v].out_neighbors[i]==w)
        {
            return nodes[v].distance[i];
        }
    }
    return 0.0f;
}

bool Path::is_in_path(int vertex)
{
    int i;

    for(i=0; i<path_length; i++) if(path[i]==vertex) return
true;

    return false;
}

void Path::add_vertex(int vertex)
{
    if(path_length>0)
path_distance+=edge_distance(path[path_length-1], vertex);
    if(nodes[vertex].isFilter()) count_filters++;
    path[path_length++]=vertex;
    securityMeasure=-1;
}

void Path::delete_from_end()
{
    if(path_length>1) path_distance-
=edge_distance(path[path_length-2], path[path_length-1]);
    if(nodes[path[path_length-1]].isFilter()) count_filters--;
    path_length--;
}

```

```

}
void Path::Print()
{
    securityMeasure=-1;
    int i;

    for(i=0; i<path_length; i++)
    {
        cout <<path[i]<<" ";
    }
}

int Path::getSecurityMeasure()
{
    // since securityMeasure takes some time to calculate, we
    // cache it
    // once calculated. value of -1 means that securityMeasure
    // was never
    // calculated for given path or that path updates
    // invalidated securityMeasure
    // so it needs to be recalculated
    if(securityMeasure==-1) recalSecurityMeasure();
    return securityMeasure;
}

void Path::markFilterCover(int vertex, bool *nodeMeasured)
{
    // algorithm: depth-first search backward (i.e. in
    // direction
    // opposing to the edge direction)

    int j, neighbor;

    for(j=0; j<nodes[vertex].in_neighbor_count ; j++)
    {
        neighbor=nodes[vertex].in_neighbors[j];
        {
            // stop if either neighbor is a filter or it's
            // already marked
            // note: for this strategy work 100% correctly, DO
            // NOT
            // mark plain path nodes before marking spoofable
            // nodes!
            if(!nodes[neighbor].isFilter() &&
            !nodeMeasured[neighbor])
            {
                nodeMeasured[neighbor]=true;
                markFilterCover(neighbor, nodeMeasured);
            }
        }
    }
}

```

```

    }
}
}
void Path::recalcSecurityMeasure()
{
    //Algorithm:
    // 1. marks all nodes that either part of the path
    // or spoofable but not filters or final node
    //
    // 2. calculate number if such nodes and return this count
as the security measure
    // (smaller means better)
    //
    bool *nodeMeasured=new bool[count];
    int i;

    //initially all nodes are "clean"
    for(i=0; i<count; i++) nodeMeasured[i]=false;

    // for all nodes (except the final node)
    for(i=0; i<path_length-1; i++)
    {
        if(nodes[path[i]].isFilter()) //if node is a filter
        {
            //find cover for this filter and mark it
            markFilterCover(path[i], nodeMeasured);
        }
    }

    // for all nodes (except the final node)
    for(i=0; i<path_length-1; i++)
    {
        if(!nodes[path[i]].isFilter()) //if node is not a filter
            nodeMeasured[path[i]]=true; //mark just this node
    }

    nodeMeasured[path[path_length-1]]=false; //make sure final
node is not marked

    //calculate securityMeasure
    securityMeasure=0;
    for(i=0; i<count; i++)
        if(nodeMeasured[i]) securityMeasure++;

    // free dynamic memory
    delete[] nodeMeasured;
}

```

```

    nodes(nodes), count(count),
    assume_unidirectional(assume_unidirectional), cb(0)
    assume_unidirectional):
    visited_from = new int[count];
};

Paths::~Paths()
{
    delete[] visited_from;
}

//implements DFS strategy
Path Paths::find_paths(int start, int finish)
{
    Path current_path(nodes, count);
    minimumPath=Path();

    find_path_between(start, finish, current_path);
    return minimumPath;
}

void Paths::find_path_between(int start, int finish, Path&
partial_path)
{
    int i, neighbor;

    partial_path.add_vertex(start);

    /* for strategy 2, stop advancing current path if it's
length already bigger than current best's length
note: this can be accomplished in the middle of building
path, i.e. we must not reach final node
to determine this condition
*/
    if( (strategy==2)
        && minimumPath.exists()
        && partial_path.getLength()>=minimumPath.getLength())
    {
        /* we stopped at this point*/
        partial_path.delete_from_end();
        return;
    }
    /* for strategy 3, apply selection strategy. note: we can
apply this strategy only if we got whole
path from start to finish built */
    if( (start==finish) && (strategy==3) &&
minimumPath.exists() )

```

```

        /* count of filter nodes*/
        int filters_cur=partial_path.getFiltersCount();
    { /* count of regular nodes (node is a regular node if and
only if it's not a filter)*/
        int regular_cur=partial_path.getLength()-filters_cur;
        /* count of filter nodes*/
        int filters_min=minimumPath.getFiltersCount();
        /* count of regular nodes (node is a regular node if and
only if it's not a filter)*/
        int regular_min=minimumPath.getLength()-filters_min;

        bool cut=
            (filters_min>0 && filters_cur==0) // prefer paths
that have at least one filter
            || (regular_cur> regular_min) // take path with
minimum of regular nodes
            || (filters_cur< filters_min); // take path with
maximum of filter nodes

        if(cut)
        {
            /* we stopped at this point*/
            partial_path.delete_from_end();
            return;
        }
    }
    if( (start==finish) && (strategy==4) &&
minimumPath.exists() )
    {
        int sec_measure_cur=partial_path.getSecurityMeasure();
        int sec_measure_min=minimumPath.getSecurityMeasure();

        if(sec_measure_cur<=sec_measure_min)
        {
            return;
        }
    }
    if(start==finish)
    {
        /* if path selection strategy applied earlier in this
function not cuts path, it's assumed to be better
than previous one
*/
        minimumPath=partial_path;

        if(strategy==1 && cb) cb(partial_path);
    }
}
#endif PRINT_ALL

```

```

        cout << " Number of
filters=" << partial_path.getFiltersCount()
        partial_path = PrintPartial_path.getLength()
        << " sec. measure=" << minimumPath.getSecurityMeasure()
        << endl;
#endif

        partial_path.delete_from_end();
        return;
    }

    /* try every edge starting current vertex */
    for(i=0; i<nodes[start].out_neighbor_count; i++)
    {
        neighbor=nodes[start].out_neighbors[i];
        if(!partial_path.is_in_path(neighbor)) /* check neighbor
node is not in the path yet */
        {
            bool skip=false;

            /* if edge has a limited bandwidth, don't go there*/
            if ((strategy>=2) &&
nodes[start].bandwidth[i]<bandwidth)
            {
                #if PRINT_ALL
                cout << "Cut because of bandwidth restrictions: ";
                //partial_path.print();
                cout << neighbor;
                cout << "
Bootleneck=" << nodes[start].bandwidth[i] << endl;
                #endif
                skip=true;
            }

            if(!skip)
            {
                find_path_between(neighbor, finish, partial_path);
            }
        }
    }

    partial_path.delete_from_end();
}

/*****

// statistics.h

#endif STATISTICS_H

```



```

#include <iostream.h>
#define STATISTICS_H
#include "paths.h"

struct HopInfo
{
    int hops;
    int filters_total;
    int regular_total;
    int count_records;

    float avg_filters;
    float avg_regular;
};

struct FilterInfo
{
    int filters;
    int hops_total;
    int count_records;

    float avg_hops;
};

struct HopInfoAvg
{
    int hops;
    float avg_filters_total;
    float avg_regular_total;
    int count_records;
    float avg_avg_filters;
    float avg_avg_regular;
    HopInfoAvg(): avg_filters_total(0.0f),
    avg_regular_total(0.0f),
    count_records(0),
    avg_avg_filters(0.0f), avg_avg_regular(0.0f) {};
};

struct FilterInfoAvg
{
    int filters;
    float avg_hops_total;
    int count_records;
    float avg_avg_hops;
    FilterInfoAvg(): avg_hops_total(0.0f), count_records(0),
    avg_avg_hops(0.0f) {};
};

const int maxSize = 100;

```

```

class Statistics
{
private:
    HopInfo hopInfo[maxSize];
    FilterInfo filterInfo[maxSize];

    HopInfoAvg hopInfoAvg[maxSize];
    FilterInfoAvg filterInfoAvg[maxSize];

public:
    Statistics();

    void startNetwork();
    void addPath(Path& path);
    void finishNetwork();
    void reportNetwork(ostream&);

    void calculateGlobalStat();
    void reportGlobalStat(ostream&);
};

#endif //STATISTICS_H

/*****

// statistics.cpp

#include "statistics.h"

Statistics::Statistics()
{
    int i;

    for (i=0; i<maxSize; i++)
    {
        hopInfoAvg[i].hops = i;

        filterInfoAvg[i].filters = i;
    }
}

void Statistics::startNetwork()
{
    int i;

    for (i=0; i<maxSize; i++)
    {
        hopInfo[i].hops = i;
        hopInfo[i].count_records = 0;
    }
}

```

```

    hopInfo[i].regular_total = 0;

hopInfo[i].filters_total = 0;
    filterInfo[i].count_records = 0;
    filterInfo[i].hops_total = 0;
}
}

void Statistics::addPath(Path& path)
{
    if(!path.exists()) return;

    int hops=path.getLength()-1;
    int filters=path.getFiltersCount();
    int regular=path.getLength()-path.getFiltersCount();

    hopInfo[hops].filters_total +=filters;
    hopInfo[hops].regular_total +=regular;
    hopInfo[hops].count_records++;

    filterInfo[filters].hops_total +=hops;
    filterInfo[filters].count_records++;
}

void Statistics::finishNetwork()
{
    int i;

    for (i=0; i<maxSize; i++)
    {
        if(hopInfo[i].count_records)
        {
            hopInfo[i].avg_filters = (float)
hopInfo[i].filters_total/hopInfo[i].count_records;
            hopInfo[i].avg_regular = (float)
hopInfo[i].regular_total/hopInfo[i].count_records;

            hopInfoAvg[i].avg_filters_total +=
hopInfo[i].avg_filters;
            hopInfoAvg[i].avg_regular_total +=
hopInfo[i].avg_regular;
            hopInfoAvg[i].count_records++;
        }
        else
        {
            hopInfo[i].avg_filters = hopInfo[i].avg_regular = 0.0;
        }
    }
}

```

```

        if(filterInfo[i].count_records)
        {
            filterInfo[i].avg_hops = (float)
filterInfo[i].hops_total/filterInfo[i].count_records;
            filterInfoAvg[i].avg_hops_total +=
filterInfo[i].avg_hops;
            filterInfoAvg[i].count_records++;
        }
        else
        {
            filterInfo[i].avg_hops = 0.0;
        }
    }
}

void Statistics::calculateGlobalStat()
{
    int i;

    for (i=0; i<maxSize; i++)
    {
        if(hopInfoAvg[i].count_records)
        {
            hopInfoAvg[i].avg_avg_filters =
hopInfoAvg[i].avg_filters_total/hopInfoAvg[i].count_records;
            hopInfoAvg[i].avg_avg_regular =
hopInfoAvg[i].avg_regular_total/hopInfoAvg[i].count_records;
        }
        else
        {
            hopInfoAvg[i].avg_avg_filters = 0.0f;
            hopInfoAvg[i].avg_avg_regular = 0.0f;
        }

        if(filterInfoAvg[i].count_records)
        {
            filterInfoAvg[i].avg_avg_hops =
filterInfoAvg[i].avg_hops_total/filterInfoAvg[i].count_records
;
        }
        else
        {
            filterInfoAvg[i].avg_avg_hops = 0.0f;
        }
    }
}

void Statistics::reportNetwork(ostream& report)
{

```

```

for(i=0; i<maxSize-1; i++)
{
    int i, j;
    j=i+1;
    if(hopInfo[j].count_records)
    {
        report <<"\tNumber of hops:\t"<<j<<endl;
        report <<"\t\tAverage
filters:\t"<<hopInfo[j].avg_filters<<endl;
        report <<"\t\tAverage
regular:\t"<<hopInfo[j].avg_regular<<endl;
    }
}

for(i=1; i<maxSize; i++)
{
    if(filterInfo[i].count_records)
    {
        report <<"\tNumber of filters:\t"<<i<<endl;
        report <<"\t\tAverage hop
count:\t"<<filterInfo[i].avg_hops<<endl;
    }
}

void Statistics::reportGlobalStat(ostream& report)
{
    int i, j;

    for(i=0; i<maxSize-1; i++)
    {
        j=i+1;
        if(hopInfoAvg[j].count_records)
        {
            report <<"Number of hops:\t"<<j<<endl;
            report <<"\tAverage
filters:\t"<<hopInfoAvg[j].avg_avg_filters<<endl;
            report <<"\tAverage
regular:\t"<<hopInfoAvg[j].avg_avg_regular<<endl;
        }
    }

    for(i=1; i<maxSize; i++)
    {
        if(filterInfoAvg[i].count_records)
        {
            report <<"Number of filters:\t"<<i<<endl;
            report <<"\tAverage hop
count:\t"<<filterInfoAvg[i].avg_avg_hops<<endl;
        }
    }
}

```

```
}  
}  
}  
/*****/
```

#2

VITA

Munirul Islam

Candidate for the Degree of

Master of Science

Thesis: REDUCING THE SCOPE OF DENIAL OF SERVICE ATTACKS
IN QUALITY OF SERVICE ROUTING NETWORKS

Major Field: Computer Science

Biographical:

Personal Data: Born in Dhaka, Bangladesh on January 20, 1976, the 2nd son
of Mahbubul Islam and Nasim Islam.

Education: Graduated from Computer Science Department, North South
University, Dhaka, Bangladesh in December, 1999; received
Bachelor of Science degree in Computer Science. Completed the
requirements for the Master of Science degree at Oklahoma State
University in May 2003.

Professional Experience: Jan 1999–Aug 1999: Teaching Assistant
Department of Computer Science, North South University, Dhaka,
Bangladesh
Oct 1999–July 2000: Junior Programmer, Devnet, Dhaka, Bangladesh
Aug 2001–Present: Graduate Assistant, CEAT Labs, Oklahoma State
University