# AN IDENTIFICATION AND ISOLATION TOOL TO ADDRESS INTERNET 2 NETWORK PROBLEMS

by

Vaibhav Prabhakar Kamath, B.E.

THESIS

Presented to the Faculty of

The University of Houston-Clear Lake

In Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

THE UNIVERSITY OF HOUSTON-CLEAR LAKE

November 2002

Copyright ©2002, Vaibhav Prabhakar Kamath

All Rights Reserved

**AN IDENTIFICATION AND ISOLATION TOOL TO ADDRESS INTERNET 2 NETWORK PROBLEMS**

by

Vaibhav Prabhakar Kamath

APPROVED BY

Dr. Sadegh Davari, Ph.D., Chair

Dr. Jim Helm, Ph.D., Committee Member

Dr. Theodore Leibfried, Ph.D., Committee Member

Dr. Robert Ferebee, Ph.D., Associate Dean

Dr. Charles W. McKay, Ed.D., Dean

To my parents, Prabha and Prabhakar, and my brother Vishal who have stood by me through all my life's endeavors.

## ACKNOWLEDGEMENTS

I am grateful for the aid and counsel of my Professors Dr. Sadegh Davari, Dr. Jim Helm & Dr. Ted Leibfried who usually without knowing it, offered inspiration to my starving love for Computer Science.

I am particularly grateful to Dr. Sadegh Davari for his indispensable assistance, endured patience and compassionate understanding of my persistent curiosity and quest for knowledge in the field of Computer Science.

It is regrettably impossible for me to mention by name all those who have encouraged, assisted or supported me, throughout my studies in the United States. To those, I would like to say thank you.

I am also indebted to my parents, Prabha and Prabhakar, and my brother, Vishal for their support, love and care.

Vaibhav P. Kamath
November 2002

ABSTRACT

**AN IDENTIFICATION AND ISOLATION TOOL TO ADDRESS INTERNET 2 NETWORK PROBLEMS**

Vaibhav P. Kamath, M.S.
The University of Houston Clear Lake, 2002

Thesis Chair: Dr. Sadegh Davari, Ph.D.

The Internet, over the years has gained tremendous popularity and has become a virtual place for people from around the world to come together breaking all possible boundaries. Many applications have been developed to make the experience on the Internet friendly. Like any new technology Internet also had some limitations. It could not satisfy the needs of applications to transfer data in real time. This led to the research and development of the Internet 2, which again has certain limitations. In this thesis we detail the limitations and try to answer some of them.

**TABLE OF CONTENTS**

## LIST OF FIGURES

CHAPTER 1

# INTRODUCTION

It is an age-old problem. Someone calls the Network Operations Center and asks whether something is wrong with the network, the reason, a particular network-based application is running slow. Where is the problem? Is it the network? Is it an overloaded server? Or, is it the client computer? Clearly, what the world really needs is a tool or technique to rapidly determine which element or elements of the system are responsible for the performance problem. [5]

In support of the Internet 2 "End to End Performance Initiative" [1], not only is a tool required to find the source of the problem but also an environment is needed on the network that would help the tool to find the source of the problem. This paper proposes to focus on the quest for one such environment.

The problem is not just with Internet 2 but is with all existing networks. There exists no such solution that can isolate the source of a slow down or break down on any given network.

The problem domain is comprised of the client machines, the server machines and the network that connects the clients and the servers, the details of which can be found in the “The End to End Performance Initiative” [1, 5]. We start off by looking for a solution to the largest of the three problem areas, the network. Once we find a solution to the network

problem further research can progress to correct and improve the software and the hardware areas on the client or the server machines.

Universities and other research organizations realized the importance and potential of having an Internet technology that was far superior to the existing Internet. A technology that could transfer high volume of information with minimum delay was one of the requirements. Internet 2 was designed as a high performance network that could transfer quality information in real time. Applications like video phones, virtual surgery, would be a reality with the Internet 2 for its real time data handling. But to date, despite having high-bandwidth, Internet 2 does not support concurrent performance. For example a video conferencing system that uses Internet 2 as the backbone for real time audio-video data transfer would not meet its benchmarks due to bottlenecks and problems at a variety of points along the network path [2].

### 1.1 The Router

Before we start our work on the quest to the solution, we must understand the basic entity of any heterogeneous network - the router. Routers work on the principle of information sharing. It consists of a computer with at least two network interface cards supporting the IP protocol (Layer 3 of the ISO model). The router receives packets from each network interface and forwards the received packets to an appropriate output network interface. Before forwarding packets on to the output interface the router must check for the Maximum Transfer Unit (MTU) of the output interface. If the received packet is larger that the MTU, it must be fragmented by the router into two or more smaller packets before

sending it on towards its destination. An important point to be noted is that a packet has a special bit called don't fragment which when set, the router will not fragment, instead discard the packet [4].

A router introduces latency as it processes packets it receives. The total delay observed is the sum of many components including:

1. Time taken to process the packet header
2. Time taken to select the correct output link (figure out the shortest or fastest path depending on the algorithm used by the router)
3. Queuing delays at the output link (happens when the link or the next router is busy)
4. Additional activities (depends on the configuration, e.g. certain routers have additional monitoring capabilities which may eat up some CPU cycles, adding to the latency)

These factors must be taken into consideration when suggesting any change to the existing network topology, including the router [3, 4].

## 1.2 Scope and Objective

In this thesis, we introduce the idea of adding special entities to the existing network, which would create an environment for any tool executed either on the client or the server to locate the source of the problem on any given network. We break the entire concept into two major parts, One, called the Measurement Infrastructure, which would allow the testing at and between points on a network segment (divide and conquer). The second, called the

Analysis tool, which would use the information from the Measurement Infrastructure to isolate and identify problems on the network.

We start of with a few basic ideas that would lead us to our desired solution to a problem that exists in far more and diverse networks than just Internet 2.

### 1.3 Outline of the Thesis

This thesis comprises nine chapters. In chapter 2, we define the problem and some basic concepts that would help in better understanding the problem; we cover possible techniques such as router to router communication, maintaining databases on the routers, modification to the traceroute program and the "error collection server" approach that were initiated in a quest to a solution. The "error collection server" approach will be researched and the next 6 chapters discuss this approach in detail.

Chapter 3 details the "error collection server" approach along with the problems and their respective solutions such as communication between various entities that form part of the approach, data maintenance, address resolution. It introduces a new entity named the "address server" that would help to maintain a single source for locating all the "error collection servers" that may exist on the network. Based on the principles of the Jini [14] technology, it helps both the analysis tools and the "error collection servers" to better communicate.

Chapter 4 proposes the hardware configuration and the detailed working of the “error collection server”. It introduces an algorithm that would help the “error collection servers” to find routers on the network, and communicate it to the “address server” to get an updated list based on similar lists maintained by the “address server” from other servers.

Chapter 5 proposes the hardware configuration and the detailed working of the “address server”. It details the data structures maintained by the address server and an algorithm to keep these data structures up-to-date for better serving the analysis tool and the “error collection servers”. It (“address server”) maintains a list of all “error collection servers” and their individual list of routers which would help in informing new “error collection servers” of duplications.

Chapter 6 explains the routing principle and the important role of forwarding packets based on the routing table concept. It addresses the important question as to how the “error collection server” traces error logs by contacting the routers in its list directly and indirectly to verify that both the routers and the link between routers are working.

Chapter 7 details the communication mechanism between the servers and the analysis tools and defines the different packet formats that the servers may use to communicate.

In Chapter 8, we briefly discuss the practical implementation of the entire process in two different directions. One approach talks of implementing the architecture in phases such

that not all errors would be caught initially. The other defines a total setup of the infrastructure before startup.

Chapter 9 concludes and summarizes the thesis, and suggests future directions related to this work.

CHAPTER 2

# OVERVIEW OF THE INITIAL ATTEMPT

The problem definition related to the "End-to-End performance initiative" was divided into many parts which included the Application on either the client or the server, the Operating System on either the client or the server, or the network that both the client and the server shared. The Network side of the problem was to assure real-time data transfer, which has not been the case. For example when experimented with sending live images of an operating room from one end of the network to the other, the images had an unacceptable delay which defeated the definition of real-time [1, 5].

## 2.1 Monitoring tools, a possible approach

In an effort to solve the abrupt lags in information transfer with the Internet 2 some propositions were made. One such proposition was to setup monitoring tools at certain points all along the network [1], which would test certain parameters of the network traffic, the use of which was yet to be decided. This would result in the determination of the cause of the problem. A problem, that exists in far more and diverse networks than just Internet 2.

## 2.2 Network concepts

Any large network is a combination of many small and heterogeneous networks interconnected by routers and bridges. These devices maintain routing tables that help quickly forward packets on a specific route. The routers use one out of a variety of

algorithms that help speed up the forwarding process [4]. At this point of time it is not clear as to whether the packets that start at one location would always take the same route to the same destination. We know that UDP packets take random routes and hence reach the destination in random order. It was initially not clear if the same concept was true for TCP, but was later learned that packets do not necessarily use the same route for the same source-destination pair, hence TCP is supposed to maintain a protocol for the proper ordering of disordered received packets.

## 2.3 Possible techniques

The five solutions that were investigated are

1. Router to sender machine communication
2. Router "store-and-forward" communication
3. Database driven error-logging routers
4. Modification to the "traceroute" program
5. The "Error Collection Server" approach

Each of these solutions is explained in the following sections.

### 2.3.1 Router to sender machine communication

On receipt of a packet force the router to send an acknowledgement to the initiator of the packet acknowledging that it was alive and will forward the packet to the next router. The implication of this idea would be an increase in the network traffic gradually leading to a collapse of the entire network. Each packet would create N number of packets, where N is

the number of hops. A small improvement would be to force an acknowledgment only if the packet processed is a probe packet (a special packet that has an identifier in its header and dummy data).

### 2.3.2 Router "store-and-forward" communication

All routers must store the source IP address of each and every packet that it forwards, thereafter waiting for a response from the next router. On receipt of an acknowledgement, discard the stored IP address; else send an error message to the initiator of the packet notifying it the address of the router that did not respond. This process would lead to the maintenance of a huge database on routers when acknowledgements are not received due to faulty network, which would eventually lead to time consumption in data access and data maintenance.

### 2.3.3 Database driven error logging routers

A database can be maintained on certain servers along the network forcing each and every router to ping every other router in its routing table, such that if a ping fails, a message would be sent to one of those servers and an entry will be made in its database. A number of precautions must be taken to ensure that extra packets are not generated. For example it is possible that two routers that have each other in their routing table may ping each other instead of only one ping being generated by either of them. The limitations to this technique is that the design and architecture of the existing routers need to be changed, which would involve a lot of investment and work for the existing networks.

### 2.3.4 Modification to the "traceroute" program

"Traceroute" is a program that when executed on the source machine lists all the routers that a packet passes through between the source and the destination. It uses ICMP [6] end back messages and the TTL (time-to-live) field in the IP header. The TTL field is an 8-bit field that the source initializes. The recommended value of TTL is specified in the Assigned Numbers RFC and is currently 64 hops.

The original idea of TTL was that it would specify a certain time span in seconds that, when exhausted, would cause the packet to be discarded. Since each router is required to subtract at least one count from the TTL field, the count is usually used to mean the number of router hops the packet is allowed before it must be discarded.

The purpose of the TTL field is to prevent packets from looping on the network infinitely. This can occur during routing transients. For example, when a router crashes or when the connection between two routers is lost, it can take the routing protocols some time to detect the lost route and find another route. During this time it is possible for the packet to end up in routing loops. Each router that receives a packet subtracts one from the count in the TTL field. When the count reaches zero, the router detecting it discards the packet and sends an Internet Control Message Protocol (ICMP) [6] message back to the originating host.

The working of Traceroute is as follows. It sends an IP packet to the destination with a TTL of one so that the first router along the path, decrements the TTL to zero and sends

back an ICMP [6] error message. This identifies the first router along the path between the source and the destination. Traceroute then sends a packet with a TTL of 2, and when the packet reaches the second router along the path the TTL becomes 0, and we get the second router using the ICMP [6] message that gets sent back. This continues until the packet reaches the destination.

In order to make use of the traceroute program for our problem domain, we can modify it such that every time traceroute sends a packet, we initialize a timer and wait for the reply just like TCP works. We may either time out or receive an ICMP [6] message. If we time out it would mean that the error had occurred between the last received router and the destination. Using this modified version of trace route even if we cannot pin point the exact location of break down, we definitely can narrow down our search. Using the last router IP and using its routing table we would be limited to a few routers in the table or the link between them.

Even though we have substantially narrowed down our search we still may not be able to use the traceroute program due to the following drawbacks. ICMP [6], the error handling protocol, is a protocol that is not supported by all existing networks. The reason, On the Internet, ping of death is a denial of service (DoS) attack caused by an attacker deliberately sending an IP packet larger than the 65,536 bytes allowed by the IP protocol. One of the features of TCP/IP is fragmentation; it allows a single IP packet to be broken down into smaller segments. In 1996, attackers began to take advantage of that feature when they found that a packet broken down into fragments could add up to more than the allowed

65,536 bytes. Many operating systems didn't know what to do when they received an oversized packet, so they froze, crashed, or rebooted.

Ping of death attacks were particularly nasty because the identity of the attacker sending the oversized packet could be easily spoofed and because the attacker didn't need to know anything about the machine they were attacking except for its IP address. By the end of 1997, operating system vendors had made patches available to avoid the ping of death. Still, many Web sites continue to block Internet Control Message Protocol (ICMP) [6] ping messages at their firewalls to prevent any future variations of this kind of denial of service attack. [7, 8]

Also, it is not necessary that the path taken between the source and the destination when a network delay was experienced will be the same when traceroute is executed. If the two paths are not the same traceroute may never report an error.

The source code to a mini version of the trace-route program to suite our error detecting needs has been listed in Appendix A.

### 2.3.5 The “Error Collection Server” approach

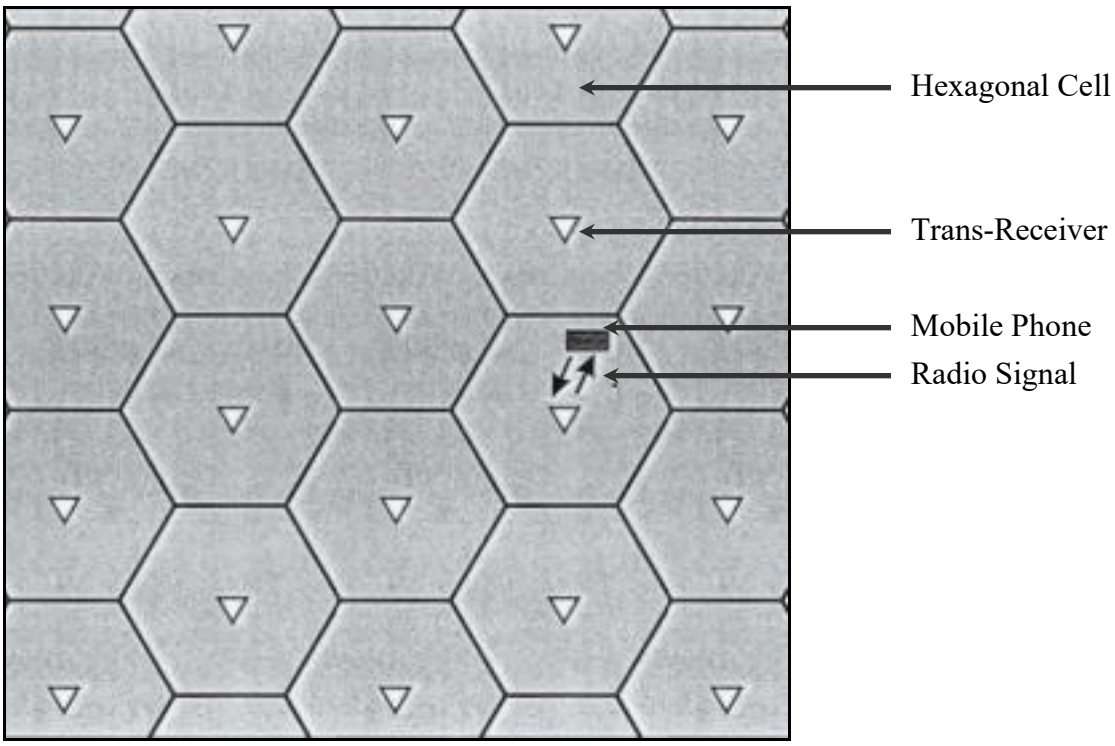



Figure 2-1: Layout of a Cell Phone Network.

A more practical solution would be to understand and implement the likes of a cell-phone network. A cell phone network has small trans-receivers installed at certain distances all over a city. These trans-receivers have a theoretical range of a hexagon, wherein the trans-receiver is at the center of the hexagon. Many such hexagons are formed adjacent to each other to cover the entire city. When a person makes a call through a cell-phone the trans-receiver of the hexagon that the person is currently in, accepts the signal. If the person is traveling and moves to the edge of the current hexagon the trans-receiver of the next hexagon the person is entering takes over. In this manner a smooth and continuous signal is ensured. A similar kind of a virtual network can be formed for the Internet, where some special servers which we may call the “error collection servers” take the place of the trans-

receivers and all the routers that fall within its virtual hexagon report to its error collection server and so on. An application, called an analysis tool, can be developed to query these servers for error locations.

The above five techniques were thought of during our quest for an over-all solution for the problem domain and are mentioned without considering or testing their practical implications.

CHAPTER 3

# THE ERROR COLLECTION SERVER APPROACH

Among the five techniques that were researched, the error collection server approach is the technique we chose for this research. This chapter will research this technique and detail the idea of implementing a virtual entity on the existing network that would form a base to our error correction system. We now explore the use of one such “error collection server”.

## 3.1 Four things to be done by the “Error Collection Server”

An error collection server should serve the following purposes.

1. Test if the group of routers assigned to it is up and running.
2. Test each and every link between each router in a group.
3. Keep a log of all faulty routers and links.
4. Supply information to Analysis tools.

In order to practically implement such a scenario we need to explore the following areas.

## 3.2 The “error collection servers” installation problem

The most important question that would cross ones mind would be as to where and how will the server fit into the existing network? The idea is to install “error collection servers” within certain proximity of routers. For higher concentration of routers more number of “error collection servers” must be allotted. Hence some “error collection servers” may

cover two or three routers and some may cover up to the maximum. What this maximum is will be decided at a later stage.

### 3.3 Communication between "error collection servers"

Assuming that these servers have been placed all over the network, what means of communication will the servers use to talk to each other, and to the analysis tool which is a software that can be run from any where on the network? A protocol needs to be designed for these "error collection servers" to communicate with each other. The information that these servers share must ensure that no two adjacent "error collection servers" end up with the same router in their list; also additional features must be added to support load sharing. The "error collection servers" must be able to share information dynamically through the communication protocol for load sharing. If one "error collection server" begins to sense that it is too busy to handle all the routers, it must have the option of requesting certain adjacent servers to share its load.

### 3.4 Error logging

How and what kind of information will these servers maintain? The idea of maintaining information on these servers is to help the analysis tool in locating the problem location by providing it with as much information as possible. A group of parameters needs to be identified for the "error collection servers" to store. A technician connected to the network querying the "error collection server" with an analysis tool may only be interested in knowing the location of the problem on the network and may not want detailed information,

for which only the required piece of information must be provided. An administrator may be interested in knowing the overall health of the network, for which case only the status of the network must be provided. A researcher on the other hand must be provided with fine details. All these criteria need to be considered when proposing the type of information that gets stored as error logs.

### 3.5 Error Collection Server address resolution problem

The “error collection servers” maintain error log. Analysis tools contact these “error collection servers” for information. How can the Analysis Tool find “error collection servers” without knowing their address? What if the “error collection server” is removed and moved to a new address? It would be easier if the existing network were designed to be flexible. An idea where in addresses are obtained dynamically must be implemented for the network to be flexible. Most of the large networks use Dynamic Host Communication Protocol (DHCP) that allows computers to obtain addresses dynamically. To extend this idea we make use of one or more servers, which we call the “address servers” having either a static IP address or a well known name such as “AddressServer.cl.uh.edu” to keep a list of all “error collection servers”. At this point of time we use only one “address server” to prevent data redundancies on the servers. The Analysis Tools may query this “address server” for potential problems. Every time an “error collection server” is connected or disconnected from the network it sends a message to the “address server” informing it of its availability. Thus all the Analysis Tool needs is the address of the “address server” through which it can retrieve the list of all “error collection servers”. With this list, the tool

can contact each and every “error collection server” directly or contact it through the “address server” depending on which implementation proves simple.

### 3.6 Router resolution problem

Each and every router must be in one of the router tables that the “error collection servers” maintain. But how can the decision be made on which routers are served by which server? Clearly an algorithm needs to be implemented to not only bind a router to a server but also to ensure that any of the routers will not be served by more than one “error collection server”.

### 3.7 Communication from Analysis tool to “error collection server”

There are two configurations for the analysis tool to retrieve information from the “error collection servers”. One, the tool can be programmed to contact the “address server” to retrieve a list of all the “error collection servers”, and then contact the “error collection servers” individually. Two, the tool can be programmed to query the “address server” that in turn will contact each and every “error collection server” to find out what errors have been logged by them. The second approach is like a search engine that accepts a query and spawns a group of “web-bots”, a small program that searches web sites for possible matches to the query.

With all these ideas the data integrity and security of neither the “address server” nor the “error collection server” must be compromised. Valuable information gets stored on these servers that makes them hack-prone. An attacker may pretend to be an “address server”

and may communicate with one or more “error collection server” and vice versa. In such a scenario data can be manipulated or deleted by the attacker. Care must be taken to ensure that no false queries are bombarded onto the servers in order to crash the servers. Some attacker may also create tons of false initial handshakes that the “error collection server” and the “address servers” perform leading to the storage of false addresses on the “address server”. Proper identification of “error collection server” must be done prior to accepting information from it.

CHAPTER 4

# ERROR COLLECTION SERVER DESIGN AND DETAIL

In this chapter, we detail the most important entity of the "error collection server approach", the "error collection server".

## 4.1 Hardware Specification

The "error collection server" just like a router can either be a hardware device or a piece of software that can be installed on any computer. Since the server has to maintain a lot of information it becomes essential for it to have a secondary storage device. At the same time, frequently accessed information such as the List of Routers cannot be stored on the secondary storage device as the access to such a device would be very slow. A faster device such as the Random Access Memory (RAM) must also be installed along with other hardware devices. RAM is much faster than a secondary storage device but cannot retain information on power loss. Data structures like routing tables that need constant updates must always be stored in the RAM and be backed up on the secondary storage devices for safe recovery from any eventuality. Data structures that do not need constant updating can be stored on the secondary storage device.

## 4.2 List of items to be stored on the server

The "error collection server" needs to store the following information.

1. List of Routers.

2. List of Adjacent “error collection servers”.
3. List of “address servers”.

### 4.2.1 List of Routers

Each and every “error collection server” should have a group of Routers that it must monitor and/or keep a log of. Hence the server must maintain a List of Routers that is under its control. It can use this list to contact and/or test any of the routers as and when required.

### 4.2.2 List of Adjacent Servers

To support load sharing, avoid duplication of router information each and every server must maintain a List of Adjacent “error collection servers”. This knowledge will not only help the “error collection server” in better load sharing but will also help it in better maintenance of the additional information through constant communication with the neighbors.

### 4.2.3 List of “address servers”

Every “error collection server” must contact an “address server” and inform it of its presence on the network at startup. In order to contact the address server every “error collection server” must keep a List of “address servers” or at least the address of one “address server”. A “list” here is mentioned keeping in mind any future upgrades when it becomes imperative to use more than one server.

### 4.3 Detailing the “error collection server”

When an “error collection server” is first introduced in the network it will be connected directly to at least one router. On startup the “error collection server” must contact the “address server” informing it of its service. Once approved the “error collection server” must find N routers which it can serve. This N is a constant and is the maximum number of routers any server can serve. No router must belong to more than one “error collection server”, which is to say that the list of routers that “error collection servers” maintain is always unique. Since the “error collection server” is not aware of which routers are already being served by other servers, the “error collection server” must search for more than N routers. In this case we consider 2N-1 routers a sufficient number. Once the server finds 2N-1 routers, it must send the list to the “address server” that maintains a list of routers that other “error collection servers” are serving, and get a reply with a new list that excludes duplicates. If the newly received list contains M routers which is less than N then the “error collection server” must change the value of N to N minus M, and then go through the process of again finding 2N -1 routers, sending the list to the “address server” and so on until it achieves the required number of routers. If the newly received list contains M routers which is equal to or more than N then the “error collection server” must truncate the list to N discarding the additional routers in the list. The objective is to find N routers distinct to all error collection servers.

Algorithm for finding N routers distinct to each “error collection server”

Set the direct router as the current router

Set two lists (temporary, permanent) to empty
Get the routing table from the current router
Go through the routing table and add all the routers directly connected to the current router to a temporary list
If the count of routers in the temporary list is equal to the 2N - 1 then
    Send the list to the "address server" and get a new list that has routers distinct to each server from it, update the permanent list
    If the count of the permanent list is not N
        Set the router at the end of the list as the current router, repeat the process
Else
    Set the router at the top of the temporary list as the current router, repeat the process

A simulation of this algorithm has been listed in Appendix A.

CHAPTER 5

# ADDRESS SERVER DESIGN AND DETAIL

In this chapter, we detail the “address server”, an entity that the “error collection server” communicates with.

## 5.1 Hardware Specification

The “address server” like the “error collection server” is another piece of hardware that must have a secondary storage device and a high configuration of Random Access Memory. The need for such a configuration is because the “address server” provides a birds-eye view of the entire network of all “error collection servers”, by maintaining a list of all active “error collection servers” and the routers that each serve. On request it sends this list of “error collection servers” to the Analysis tool.

## 5.2 List of items to be stored on the server

The “address server” needs to store the following information.

1. List of “Error Collection Servers” with their list of routers
2. List of other “Address Servers” (for future use, initially this list will be only one row long)

### 5.2.1 List of “Error Collection Servers”

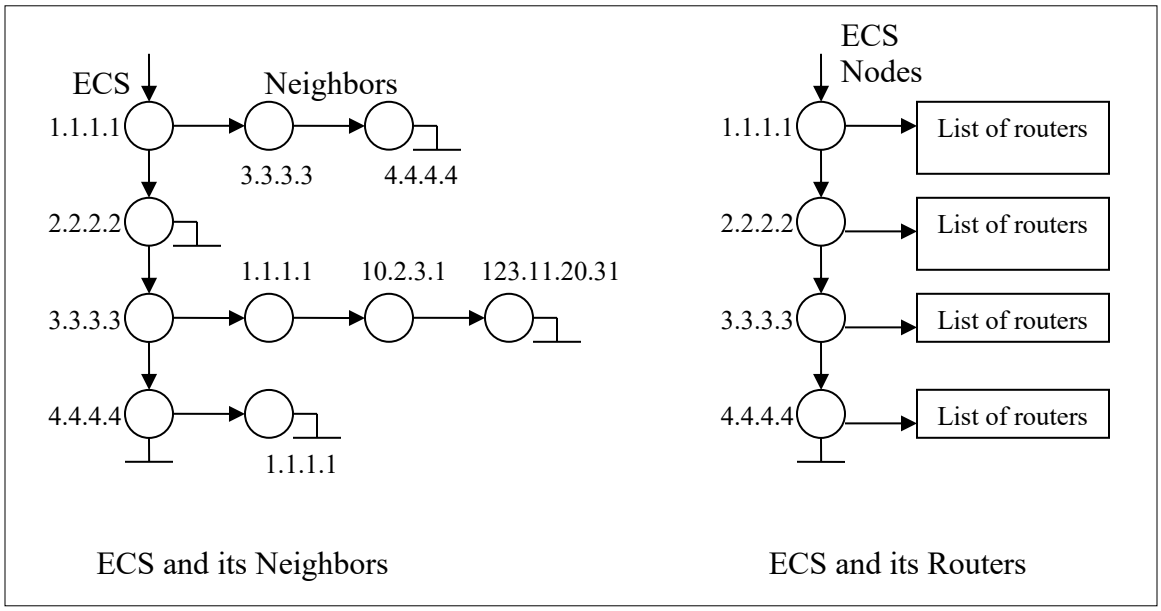
Every time an “error collection server” boots up it must contact the “address server” to inform it of its presence on the network. The “address server” must scan through its list of “error collection servers” to check for duplicate and approve the “error collection server” to serve, at the same time make an entry in the list for the “error collection server” along with the time it was contacted. The “address server” must receive a list of routers from the “error collection server” that it must maintain to assist the “error collection servers” in load sharing. The “address server” at any given time can import the list of routers from any “error collection server” using the address it maintains. The amount of information stored with the “error collection server” such as list of routers for each “error collection server” can be used to create a virtual map of the entire network, which would help an administrator to better understand the network.

### 5.2.2 List of other “Address Servers”

The current configuration has only one “address server” that handles all “error collection servers”. More “address servers” may be added to the system as future enhancements that would help in distributing the load between “address servers”.

## 5.3 Detailing the Address Server

The “address server” must maintain at least two lists in the form of linked lists detailing the “error collection server”. The first list must maintain all “error collection servers” and their neighbors, the second must maintain all “error collection servers” with their routers.




The above diagram is a snapshot of the data structures that the “address server” maintains. Everytime an “error collection server” contacts the “address server” it scans through the “ECS and its neighbors” list to check if an entry for it already exists. If there is no entry it approves the “error collection server” to serve and adds it to the list. On reciept of the list of routers and their modified routing tables the “address server” scans the routers through the “ECS and its Routers” list to check for existance. If the router exists in the list of some other “error collection server” it is marked as remove on the newly received list, since no two “error collection servers” can serve the same router. Using this process when the “error collection server” sends a final list that is free of any duplications, the “address server” adds the new “error collection server” and its list of routers to the “ECS and its Routers” list. It then scans through the individual routers modified routing tables to check for neighbors. For example if a router 121.23.41.2 that is associated with “error collection server” 1.1.1.1 exists in the routing table of one of the routers associated with “error

collection server" 4.4.4.4 the two "error collection servers" are neighbors, and an update must be made to the "ECS and its neighbors" list.

Thus for every new "error collection server", the "address server" must follow the above steps. The two lists maintained on the "address server" can be retrieved to any computer and can be used to create a virtual mapping of the network topology.

Algorithm for maintaining these lists

```
On receipt of data from "error collection server"
Separate the "error collection server" information and the list of routers.
Loop through each Node on the list of neighbors
        If Node already exists
                Check for changes in the list of routers and update it
        Else Create a New Node and append it to the end of the list
                Search for potential neighbors by comparing the routers in the new
                node and the routers in existing nodes.
                If match found update the list of neighbors and mark the router as
                'remove' on the new node
```

The reason we mark the router as remove in the new node is because a match indicates that the router is already being served by some other "error collection server" hence notify the new "error collection server" to remove the router from its list. With the help of this algorithm we can picturize the setup as it progresses.

A simulation of this algorithm has been listed in Appendix A.

CHAPTER 6

# THE ROUTING PRINCIPLE

The most important role played by the network is the forwarding of packets based on a table that it maintains. This table known as the Routing table is the information source for the router to know if the selected output link is directly or indirectly connected to the destination. The initial few entries to the routing table is made manually by the administrator of the network to which the router is connected. Although the information in the Routing Table changes periodically, there are at least two common columns to all routers that rarely change.

1. Network address or name of a network (this is the address of the next router that may lead to the destination)
2. Default route (used as the output link if none of the other entries lead to the destination for a packet)

## 6.1 IP address or network name

All routers maintain a list of IP addresses; each being a direct link to another interface in the network to which it can forward packets. Routers make use of the routing protocol to communicate and exchange routing information, like a change in a routing table.

## 6.2 Default route

Each and every router maintains at least one default route which is a link to an interface on which it would forward packets in case none of the other interfaces led to the destination.

The value in the default route column of the routing table may be “direct” if the router is directly connected to the destination network or the IP address of the router directly connected to it on the default interface. Using this piece of information from all routers we can create a virtual image (a graph) of the entire network.

Every time the “error collection server” boots up it should contact the routers in its list and request them for their routing table.

With the help of the routing table from all routers in its list it can create a tree that indicates routers that are directly connected to each other.

The details are as follows

Let us say that we have the four routers in the List of Routers

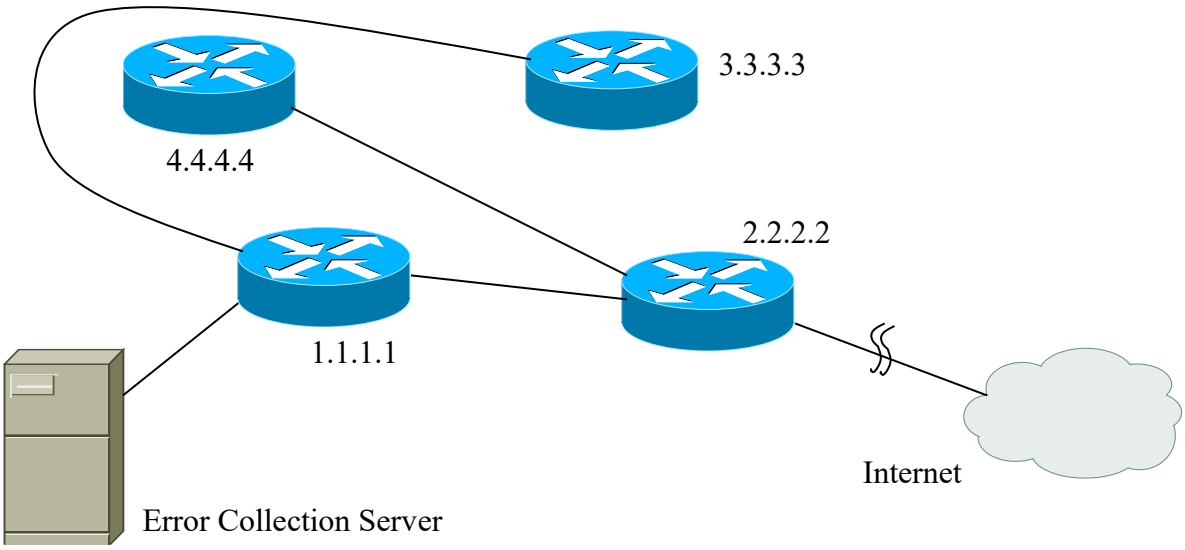



Figure 6-1: Error collection server connected to the network.

We receive the following Routing Tables from them.

| Network | Gateway |
|---|---|
| 2.2.4.35 | 2.2.2.2 |
| 3.4.5.6 | 3.3.3.3 |
| Default | 2.2.2.2 |

Router 1.1.1.1

| Network | Gateway |
|---|---|
| 1.1.2.2 | 1.1.1.1 |
| Default | 1.1.1.1 |

Router 3.3.3.3

| Network | Gateway |
|---|---|
| 1.2.3.4 | 1.1.1.1 |
| 4.3.3.2 | 4.4.4.4 |
| Default | 5.6.7.8 |

Router 2.2.2.2

| Network | Gateway |
|---|---|
| 2.3.4.5 | 2.2.2.2 |
| Default | 2.2.2.2 |

Router 4.4.4.4

We can imagine the list of routers maintained in an array as follows.

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 1.1.1.1 | 2.2.2.2 | 3.3.3.3 | 4.4.4.4 |

A two dimensional array can be created with the help of the above information as follows

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 (node itself) | 10 (seconds) | 25 (seconds) | ∞ |
| 2 | 10 (seconds) | 0 (node itself) | ∞ | 30 (seconds) |
| 3 | 25 (seconds) | ∞ | 0 (node itself) | ∞ |
| 4 | ∞ | 30 (seconds) | ∞ | 0 (node itself) |

For all those routers that have a direct link a positive value such as the time delay for a message exchange can be inserted, for all those routers that do not have a direct link an infinite value can be inserted and for the node itself a zero can be inserted.

A copy of the above chart can be used as a reference for the “error collection server” to mark all those routers and links that the “error collection server” has finished processing. This would eliminate any duplicate processing.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | X | X | 0 |
| 2 | X | 0 | 0 | - |
| 3 | X | 0 | 0 | 0 |
| 4 | 0 | - | 0 | 0 |

0 – Self node / no connection

X – Processed

- – Yet to be processed

It is necessary to keep in mind that the design of the “error collection server” must be such that no router in the entire network is left out i.e. to say that each and every router is part of at least one “error collection server”.

The following are the parameters that the “error collection server” may store as part of the error log.

1. Hop count.
2. Time taken.

3. From address. This is the address(s) of all the routers that were entered as a part of loose source routing.
4. To address.
5. Status. This can be a variety of items like slow, down …
6. If possible the theoretical values of the network like bandwidth based upon the theoretical value for ‘time taken’ can be deduced. (Optional).

Every time the “error collection server” starts up, it must contact the “address server” to inform it that it is ready to serve. In return the “error collection server” must receive a message from the “address server” of its acceptance along with the total amount of time it has within which it must renew its service. This is especially necessary for the “address servers”, as it would help them in keeping track of all the “error collection servers” that are alive and those that may have crashed. More details can be thought about as this idea is based on the principles of the Jini [14] Technology design.

# CHAPTER 7

# COMMUNICATION DETAILS

We have seen the details of the hardware design for the “error collection server” and the “address server”. We now present how these servers communicate among themselves and with each other.

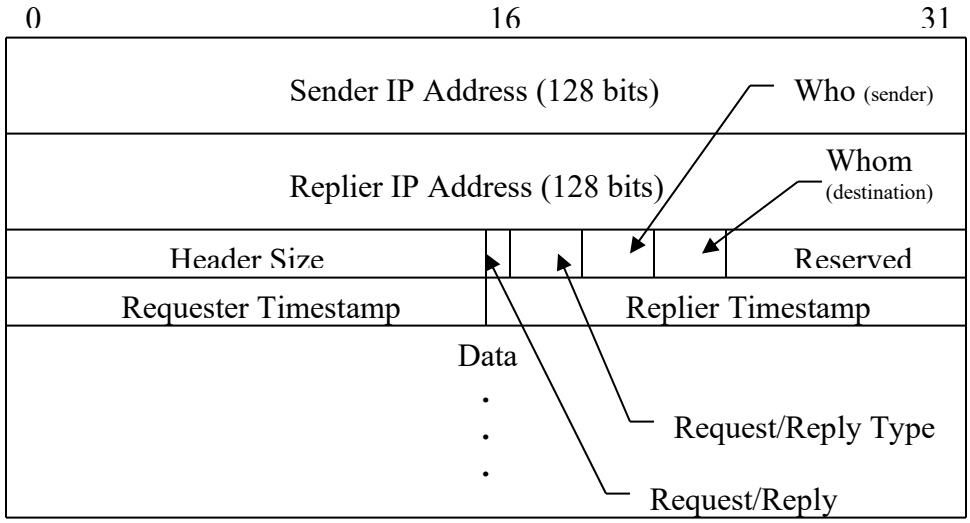
## 7.1 Communication between “error collection servers”

Each “error collection server” maintains three lists: a list of Routers, a list of adjacent “error collection servers”, a list of “address servers”. The “error collection server” may talk to another “error collection server”, an “address server” or an Analysis Tool.

We build our Communication Protocol above the TCP/IP package so that we can let TCP do all the error handling.

The basic component in any communication is a Data Packet. So, we detail a Data Packet that forms a part of “error collection server” to “error collection server” communication.

A Data packet for “error collection server” may look as follows




**Header Size:** This field is used to calculate the position in the packet from which Data starts.

**Request/Reply:** This field is used to inform the receiver if the current packet is a request or a reply to a previous request from it.

0 – Request.

1 – Reply.

**Request/Reply Type:** An “error collection server” may request another “error collection server” for its list of Routers, List of “address servers” or for Load Sharing. These request are for the “error collection server” to test whether it is up-to-date. The list of routers will help the “error collection server” to check for duplications in its list. The list of “address servers” will help the “error collection server” to check for additional “address servers” in

the network, which it may not have. The request for load sharing will help in easing the load on this “error collection server” by transferring some of the load to the “error collection server” that accepted its request for load sharing. Finally the Accept Load will inform the other “error collection server” that the Data field contains the list of routers that it may take control of. The sender “error collection server” will then wait for a Reply with the Reply field as 1 (Yes) before it frees control of the routers it sent to the other “error collection server”.

If the Request/Reply field is 0 (Request), the data in this field may be interpreted as follows.

0 – List of Routers.

1 – List of “address servers”.

2 – Load Sharing.

3 – Accept Load.

If the Request/Reply field is 1 (Reply), the data in this field may be interpreted as follows.

0 – No.

1 – Yes.

2 – Busy.

3 – Error.

**Who:** Since the design of most of the packets that are transferred between “error collection servers”, “address servers” and Analysis Tools are almost the same, we use the fields “Who” and “Whom” to identify who sent the packet and to whom. This way if the packet

sent to a “error collection server” reaches an “address server”, based on the Whom field, the “address server” will know for sure whether it was meant for it or not.

**Who:** Identifies the originator of this packet.

0 – “error collection server”.

1 – “address server”.

2 – Analysis Tool.

3 – Don’t Care.

**Whom:** Identifies the destination of this packet.

0 – “error collection server”.

1 – “address server”.

2 – Analysis Tool.

3 – Don’t Care.

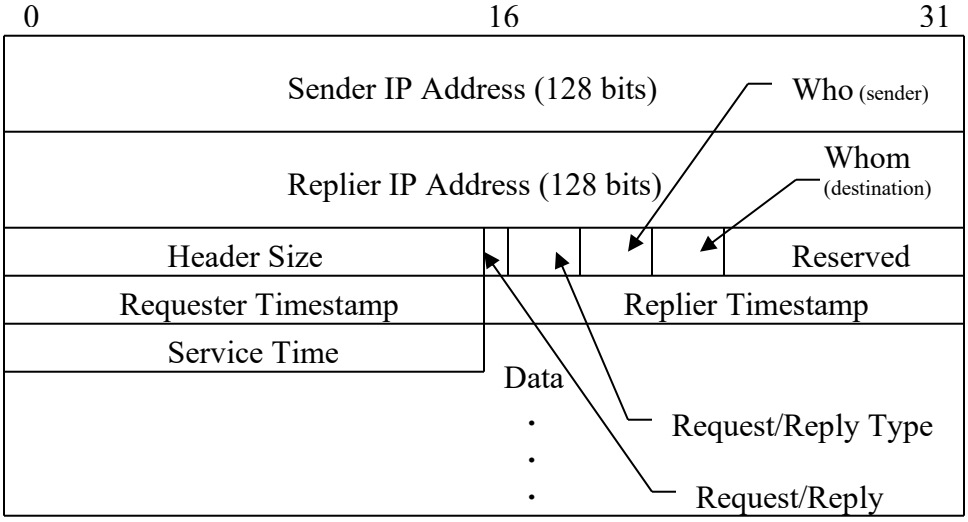
**Requester Timestamp & Replier Timestamp:** These fields are used just to keep track of the time when the request was initiated and when it was executed.

**Data:** The data field is optional and is filled by the replier based on the request field. For example if the requester set the request field to 0 the replier fills this data field with all the routers it has in its list. On the other hand if the request field is set to 2 the replier will set only the reply field and ignore the data field.

## 7.2 “Error collection server” to “address server” communication

Every “error collection server” must contact the “address server” to inform it of its desire to serve. This is done when the “error collection server” boots up. When contacted the “address server” adds the “error collection server” to its list and gives it a time frame within which it must renew its lease with the “address server”. This way if the “error collection server” does not renew its lease the “address server” will know that either the “error collection server” has crashed or it is no more interested in serving. This information goes back and forth in the form of a packet. So, let us detail a Data Packet that forms a part of “error collection server” to “address server” communication.

A typical “error collection server” to “address server” packet may look as follows



**Request/Reply Type:** An “error collection server” may request the “address server” to add it to its list and allow it to serve. If it wants to renew the lease it may also send a renewal request. In case of an overload it may contact the “address server” for it to see if it can find another “error collection server” that can share the load.

If the Request/Reply field is 0 (Request), the data in this field may be interpreted as follows.

0 – New “error collection server”.

1 – Renew “error collection server”.

2 –Overload.

3 – Don’t Care.

If the Request/Reply field is 1 (Reply), the data in this field may be interpreted as follows.

0 – Error.

1 – Ok.

2 – Busy.

3 – Don’t Care.

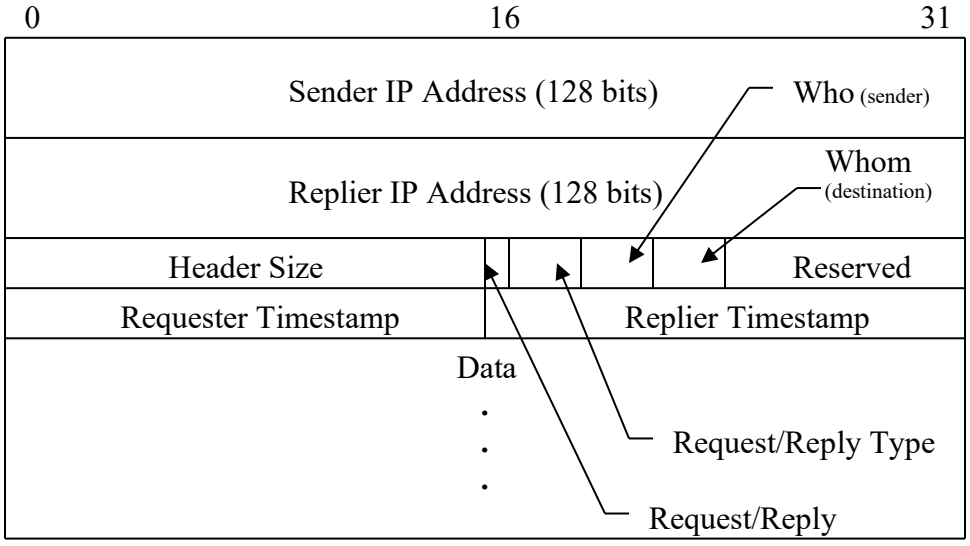
**Service Time:** This field is field by the “address server” when it gets 0 (New “error collection server”) or 1 (Renew “error collection server”) in the Request Type. It indicates the amount of time the “error collection server” has to renew its lease with the “address server”.

**Data:** The data field is optional and may be used only when the "address server" responds to a Request Type of 3 (Overload). It fills the Data field with a list of "error collection servers" near the requesting "error collection server" that it thinks may help share its load. All other fields remain the same as that of the "error collection server" to "error collection server" Communication packet.

### 7.3 Analysis Tool to "address server" communication

An Analysis Tool run from any computer on the network may contact the "address server" to request the list of "error collection servers" it may contact to find the exact source of the problem. This information goes back and forth in the form of a packet. So, let us detail a Data Packet that forms a part of Analysis Tool to "address server" communication.

A typical Analysis Tool to "address server" packet may look as follows



**Request/Reply Type:** All that the Analysis Tool requests from an “address server” is a list of “error collection servers”.

If the Request/Reply field is 0 (Request), the data in this field may be interpreted as follows.

0 – List of “error collection servers”.

1 – Don’t Care.

2 – Don’t Care.

3 – Don’t Care.

If the Request/Reply field is 1 (Reply), the data in this field may be interpreted as follows.

0 – Error.

1 – Ok.

2 – Busy.

3 – Don’t Care.

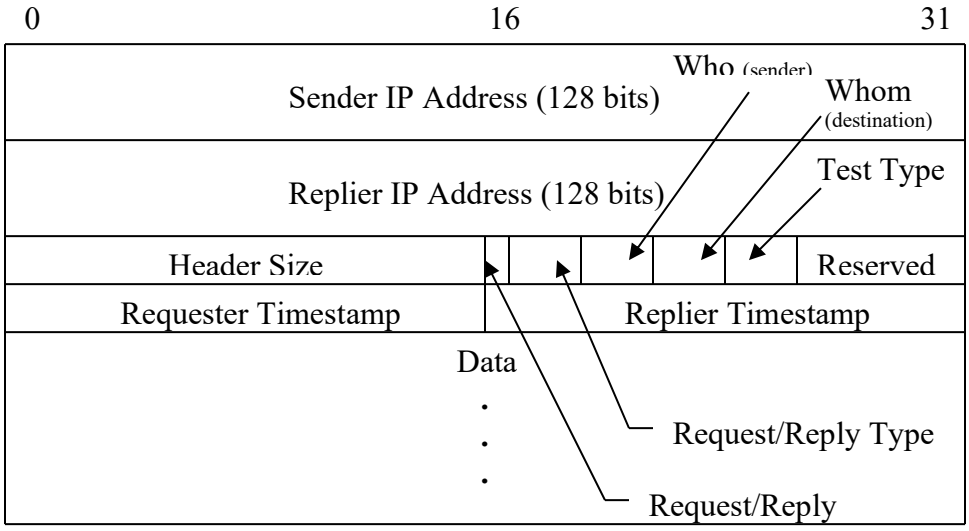
**Data:** “address server” responds to a Request Type of 0 (List of “error collection servers”) and fills the Data field with a list of “error collection servers”.

All other fields remain the same as that of the “error collection server” to “error collection server” Communication packet.

### 7.4 Analysis Tool to “error collection server” communication

An Analysis Tool contacts an “error collection server” to find if it detected an error in any of the paths between the routers that it has in its list. The severity of this test may depend on the Analysis Tool. This information goes back and forth in the form of a packet. So, let us detail a Data Packet that forms a part of Analysis Tool to “error collection server” communication.

A typical Analysis Tool to “address server” packet may look as follows




**Request/Reply Type:** The Analysis Tool requests from an “error collection server” either the status of the network or the list of routers or their links that have problems. If it only wants the results from the log that the “error collection server” maintains it may set the Type as 0. If it wants the “error collection server” to initialize a fresh test it may set the field to 1.

If the Request/Reply field is 0 (Request), the data in this field may be interpreted as follows.

0 – Test.

1 – Thorough Test.

2 – Don’t Care.

3 – Don’t Care.

If the Request/Reply field is 1 (Reply), the data in this field may be interpreted as follows.

0 – Error.

1 – Ok.

2 – Busy.

3 – Down.

**Test Type:** This is used to indicate the level of response the Analysis Tool wants from the “error collection server”.

0 – Status.

1 – List of IP addresses that have errors.

2 – Details.

3 – Don’t Care.

**Data:** The “error collection server” responds to a Test Type of 1 (List of IP addresses that have errors) and fills the Data field with a list of Router IP addresses that was unreachable. In case of Test Type 2 (Details) it embeds a series of messages that contain the faulty IP address, the time when the test was done, the total hops taken to reach the faulty IP address, and other details that the “error collection server” may have.

**All other fields remain the same as that of the “error collection server” to “error collection server” Communication packet.**

CHAPTER 8

# THE FINAL SETUP

So far we have seen the theoretical detail of how we can solve the End-to-End Performance problem. We now go a step forward and look at the practical implementation.

When we think about the problem and the details we have learnt so far the following questions arise.

**What should be the initial network setup that would help us in achieving our final objective?**

Probable solution

An “address server” can be setup anywhere along the network and be publicized on the Internet so that all the administrators can easily get its address.

The “error collection servers” can be initialized all at once or can be setup one at a time.

This idea is similar to the floodlights in a football field where one switch sets all the bulbs on at the same time or as in a theatre where the bulbs or a group of bulbs go on one at a time.

## 8.1 The all at once approach

In this technique the “error collection servers” may contact the “address server” and wait until they receive a go ahead from the “address server”. The “address server” on the other hand waits for as many contacts from “error collection servers” until it feels that the count is enough to serve all the routers in the Internet 2 Architecture. It may assume the value of

the count by using the initial list of Routers (manually feed by the administrator of those individual networks) of each “error collection server” and by using probability theory to foresee the outcome of the probe (A probing algorithm may be implemented) initialized by each “error collection server” such that this new probable list of routers of each “error collection server” covers almost the entire network.

This idea assumes the entire network of the “error collection servers” to be up before the process starts working, which means it could be years before we get a working model in place.

### 8.2 The one at a time approach

Here the administrator manually feeds a group of routers known to him (from his own network) into the “error collection server”. On startup the “error collection server” first contacts the “address server” to inform it of its readiness to serve, at this point the “address server” stores the new entry into its list of “error collection servers”. Next, the “error collection server” starts probing for other possible routers that it may include using the current list of Routers; it may probe until it sees that it is overloading itself or until it reaches a maximum limit (A default value which may be factory-set into it, configurable later by an administrator). How these “error collection servers” are going to solve conflicts (duplication of part of its list of Routers) is explained later. Thus when the practical implementation of the entire idea comes to life only a few “error collection servers” may be serving a few distant set of routers hence not all problems may be caught at the beginning phase. As “error collection servers” go up one by one, more and more area on the network will be covered leading to more and more errors being caught by the system

until all the routers on the entire Internet 2 architecture are part of one “error collection server system”.

**How does the “error collection server” find the adjacent routers and add them to its list?**

To help the “error collection server” find routers a probing algorithm may be implemented which search the network for unused routers (those routers that do not appear in any “error collection servers” list.

The basis to this algorithm is as follows

For each router in the List of Routers

The “error collection server” requests the router for its Routing table.

Deduces all the routers that have a direct connection to the above router and adds them to its list.

Follow the above steps for the newly added routers.

It is important to note that the above process may loop infinitely. As each router has at-least one direct connection, hence the probe must end as soon as it reaches a maximum level.

**How does the “error collection server” find the adjacent “error collection servers” and add them to its list?**

In order to find Adjacent “error collection servers” for each “error collection server” the following idea may also be implemented.

Every time an “error collection server” contacts the “address server” for the first time.

The “address server” performs the following tasks.

It looks up for the address of the new “error collection server” in its list of “error collection servers”, if found, sends an error message, else adds the new address to its list.

Sends the address of the newly added “error collection server” to all the other “error collection servers”.

Every “error collection server” that receives the address of another “error collection server” from the “address server” executes an algorithm that tests to see if this “error collection server” is its neighbor. If yes adds to its list of Adjacent “error collection servers”, sends a message to the newly added “error collection server” informing that it is its neighbor so that the other “error collection server” may update its list. Thus for every new entry other “error collection servers” will update their own list and contact the new entrant to add its new neighbor.

Chapter 9

# CONCLUSION

In this Thesis, we have introduced a new Measurement Infrastructure called the error collection server approach that provides an interface for the end user to query an infrastructure for the exact location of a problem on the network. The approach uses a combination of entities like “error collection server”, “address server”, and “analysis tool” designed for error identification and isolation.

We have investigated error detection techniques for the network that would obtain a solution to the problem domain. Five methods have been investigated in chapter two, namely Router to sender communication, router store-and-forward communication, data driven error-logging routers, modification to the trace route program and the error collection server approach. Each of these methods had their drawbacks which were detailed and discussed. The first three methods required modification to the working of the routers which meant that the system would not work until all existing routers comply to the change, they also had the risk of increasing the network traffic that could lead to a network breakdown. The modification to the trace route program looked promising but had certain drawbacks that would limit the use of the program to networks that supported ICMP [6], a protocol that is disabled by many networks. The “error collection approach” served as both an error locator and a network monitor and hence was chosen for further research.

The approach works on the principle of using small mini computers that could be setup all across the network and would serve two purposes. One, it would search its surrounding

routers for possible faults and two; it would serve as a network monitor. In searching for errors the response time can be used to deduce the average time and hence the average speed at which information got exchanged. Information like the average speed and network traffic can prove helpful in identifying congestion and deadlocks. It would help in better understanding network behavior and can lead to designing smart algorithms for routers to divert traffic using this information.

Algorithms for the “error collection server” to search for routers and for the “address server” to maintain information obtained from the “error collection server” have been designed and detailed. These algorithms implement data manipulation based on linked lists. A sequential search is required for an item in the list which can be changed to quick parallel search as a future enhancement. The communication protocol for information exchange between the three entities namely the “error collection server”, “address server” and the “analysis tool” have been explained and the smallest entity of any communication, the “packet” has been detailed. One of the important enhancements that need to be addressed is the security model for the servers. Although the servers use a custom communication protocol, an attacker can pretend to be any of the servers and initiate a communication with malicious intent of disrupting communication with other servers. The attacker may spam the server with false information in order to crash the server. Certain secure protocols may be developed as an extension to this work.

The “error collection server” at this time communicates only with one “address server”. As the network grows the load on the “address server” would increase and a set of other

“address servers” would be required. The addition of “address servers” must however consider the following hurdles. At any given time all the “address servers” need to be up to date, lacking which will defeat the entire purpose. More “address servers” would mean the requirement to communicate the addresses of them to the “error collection servers” and the “analysis tools”. Algorithms for load sharing and load balancing for the “address servers” will have to be implemented to reduce load and hence congestion on certain servers. These algorithms must also be carried over to the “error collection server” as certain servers may get overloaded in which case they must be able to request neighboring servers for load sharing.

Although this concept was designed for a network it can be used in a variety of other areas such as fault tolerance, telecommunications, etc. Implementing the techniques mentioned in this thesis can be difficult on architecture as huge as Internet or Internet 2 but it can surely be implemented and tested on a smaller scale on intranets.

We suggest the above details be considered as part of an extension to this work.

## APPENDIX A

A Mini Trace Route Program

The algorithm

Initialize a dummy data packet and the output UDP socket.
Increment the TTL field in the outgoing packet one by one sending the dummy data to the destination.
Wait for a reply
If the reply is not received within a given time, there could have been an error along the path.
    Display the error message.
Else
    Display the destination IP address along with the time taken for the response.
Continue this process until the TTL reaches a maximum (30) set by the user.

The source code

```
/*****************************************************************
*      TraceRoute.c                                              *
*      This program is a mini version of TraceRoute that waits for *
*      an ICMP Response from a node along the destination path and *
*      tries to detect those nodes that may be dead              *
*****************************************************************/

#include <netinet/in_systm.h>
#include <netinet/ip_icmp.h>
#include <netinet/udp.h>
#include <netinet/in.h>
#include <netinet/ip.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/ipc.h>

#include <arpa/inet.h>

#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define DATASIZE    255
#define UDP_PORT    9887         // Choose a high Port # that is rarely used.

#define MINTTL      1
#define MAXTTL      30
#define WAITTIME    3

#define NONE (fd_set *) NULL
#define NEVER (struct timeval *) NULL
```

```
#define NPROBES     3      // Probing the destination N times to strengthen
                           // the probability of it staying alive.

struct DummyPacket {       // This is the actual probe packet format
       struct ip Ip;
       struct udphdr Udp;
       int SequenceNumber; // This is used to sort packets at the destination
       int TimeToLive;     // Number of hops the packet will survive
       struct timeval TimeStamp;  // Out going time for the packet
};

int SockID;
char *Data;
struct DummyPacket SendPacket;    // Out going packet
struct sockaddr_in dest_addr;

void Usage() {
       printf("USAGE: traceroute hostname\ne.g. traceroute www.cl.uh.edu\n");
       exit(1);
}



float TimeDifference(struct timeval *StartTime, struct timeval *EndTime) {
       //
       // Multiply 1000 to get the result in Milli Seconds
       //
       return((EndTime->tv_sec - StartTime->tv_sec) * 1000);
}

int InitializeUDP(char *DestAddr) {
       struct protoent *protocol;
       struct sockaddr_in cli_addr;

       //
       // Get the destination address
       //
       bzero((char *) &dest_addr, sizeof(dest_addr));
       dest_addr.sin_family = AF_INET;
       dest_addr.sin_addr.s_addr = inet_addr(DestAddr);
       dest_addr.sin_port = htons(UDP_PORT);

       if ((protocol = getprotobyname("icmp")) == NULL) {
              printf(stderr, "ICMP: unknown protocol\n");
              exit(2);
       }

       if((SockID = socket(AF_INET, SOCK_DGRAM, protocol->p_proto)) <0) {
              printf("TRACEROUTE: can't open datagram socket\n");
              exit(3);
       }

       //
       // Get my address
       //
       bzero((char *) &cli_addr, sizeof(cli_addr));
       cli_addr.sin_family = AF_INET;
       cli_addr.sin_addr.s_addr = htons(INADDR_ANY);
       cli_addr.sin_port = htons(0);

       if(bind(SockID, (struct sockaddr *) &cli_addr, sizeof(cli_addr)) < 0) {
              printf("TRACEROUTE: can't bind local address\n");
              exit(2);
```

```
        }

        //
        // Initialize Probe Packet
        //
        bzero((char *) &SendPacket, sizeof(SendPacket));
        SendPacket.ip.ip_dst = dest_addr.sin_addr;
        SendPacket.ip.ip_tos = 0;
        SendPacket.ip.ip_v = IPVERSION;
        SendPacket.ip.ip_id = 0;

        //
        //     Request some space for the dummy data
        //
        Data = (char *)malloc(DATASIZE);
}

int ReadUDPData() {
        int Read;

        Read = recvfrom(SockID, Data, DATASIZE, 0,
                              (struct sockaddr *)0, (int *)0);
        if(Read < 0) {
                printf("TRACEROUTE: recvfrom error\n");
                exit(3);
        }
        Data[Read] = '\0';

        return Read;
}

void WriteToDestination(SequenceNumber, TimeToLive) {
        int Sent, Len, DataLen;

        SendPacket.SequenceNumber = SequenceNumber;
        SendPacket.TimeToLive = TimeToLive;

        Len = sizeof(dest_addr);
        DataLen = strlen(Data);
        if((Sent = sendto(SockID, (char *)SendPacket, DataLen, 0,
                              (struct sockaddr *)&dest_addr, Len)) != DataLen) {
                printf("TRACEROUTE: sendto error on socket, " +
                                              "only %d bytes sent\n", Sent);
                exit(3);
        }
}

int WaitForResponse() {
        struct timeval WaitTime;
        fd_set ResponseDescriptor;

        WaitTime.tv_sec = WAITTIME;
        FD_ZERO(&ResponseDescriptor);
        FD_SET(SockID, &ResponseDescriptor);

        if(select(FD_SETSIZE, &ResponseDescriptor, NONE, NONE, &WaitTime) < 0) {
                printf("TRACEROUTE: select error ...\n");
                exit(3);
        }

        if(FD_ISSET(SockID, &ResponseDescriptor))
                return ReadUDPData();
```

```
	return 0;
}

int main(int argc, char *argv[]) {
	struct timezone TimeZone;
	struct timeval StartTime, EndTime;
	char *Data, Token[DATASIZE], FileName[FILESIZE];
	int Value, TimeToLive, Probe, SequenceNumber, Received;

	if (argc < 1)
		Usage();

	SequenceNumber = 0;
	InitializeUDP(argv[1]);

	if((Data = (char *)malloc(DATASIZE)) == NULL) {
		printf("Cannot allocate Memory....\n");
		printf("Terminating program....\n");
		exit(1);
	}

	for(TimeToLive = MINTTL; TimeToLive <= MAXTTL; TimeToLive++) {
		printf("%2d", TimeToLive);

		for(Probe = 1; Probe <= NPROBES; Probe++) {
			//
			// Get the time before sending a packet
			//
			gettimeofday(&StartTime, &TimeZone);
			//
			// Send the Packet
			//
			WriteToDestination(SequenceNumber, TimeToLive);
			//
			// Wait for the Response
			// Use Select to wait for multiple sources of inputs
			//
			while(Received = WaitForResponse()) {
				//
				// Get the time after recieving a reply
				//
				gettimeofday(&EndTime, &TimeZone);
				printf("%f ms", TimeDifference(&StartTime, &EndTime);
			}
			//
			// The received data can be further decoded to check for
			// specific reasons if the packet did not reach the
			// destination, A '*' here means error occurred.
			//
			if(Received == 0)
				printf(" *");
		}
	}

	close(SockID);
	return 0;
}
```

## The Error Collection Server Simulation

The simulation consists of a TCP socket based server that works as an address server and listens of an assigned port for clients that play the Error Collection Server. The client reads a list of routers from a file and sends it to the server, which performs a lookup for duplication in list, which it maintains to keep track of Error Collection Servers and their routers.

Compilation process for the Server

gcc AddressServer.c ErrorExit.c PassiveSock.c –o AS.out –lsocket –lnsl

This creates an executable file named AS.out, next execute the following

AS.out 9887

This statement starts the server on port 9887

Compilation process for the Client

gcc ErrorCollectionServer.c ErrorExit.c –o ECS.out –lsocket –lnsl

This creates an executable file named ECS.out, next execute the following

ECS.out localhost

This statement looks for a server on the local system or the system specified as a parameter.

Changing the values in the ECSConfig.txt before running the client will result in the client sending different set of routers to the server.

```
/*****************************************************************
*      AddressServer.c                                           *
*      This program is a Server & talks to the clients namely    *
*      the Error Collection Server using Socket on TCP port.     *
*****************************************************************/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <stdio.h>

#include "LinkedList.h"

extern int    errno;
extern char   *sys_errlist[];

#define IPADDRLEN   20
#define MAXROUTERS  100
#define ARRAYLEN    IPADDRLEN * MAXROUTERS + 10

#define FALSE 0
#define TRUE  1

#define QLEN  5

struct node *routers;

char *service;
int noOfRouters;
int msock, size, type;

/*     Writes N bytes to the output stream specified  */
int WriteN(register int fd, register char *ptr, register int nbytes) {
       int    nleft, nwritten;

       nleft = nbytes;
       while (nleft > 0) {
              nwritten = write(fd, ptr, nleft);
              if (nwritten <= 0)
                     return(nwritten);          /* error */

              nleft -= nwritten;
              ptr   += nwritten;
       }

       return(nbytes - nleft);
}

/*     Writes N bytes to the output stream specified along with a new line
       character     */
int WriteLine(int fd, char *ptr, int nbytes) {
       ptr += nbytes;

       if (*(ptr-1) != '\n') {
              *ptr = '\n';
              ptr -= nbytes++;
       }
```

```
	WriteN(fd, ptr, nbytes);
}

/*	Read a line from the specified stream	*/
int ReadLine(register int fd, register char *ptr, register int maxlen) {
	int	n, rc;
	char	c;

	for (n = 1; n < maxlen; n++) {
		if ( (rc = read(fd, &c, 1)) == 1) {
			if (c == '\n')
				break;
			*ptr++ = c;
		} else if (rc == 0) {
			if (n == 1)
				return(0);	/* EOF, no data read */
			else
				break;		/* EOF, some data was read */
		} else
			return(-1);	/* error */
	}

	*ptr = 0;
	return(n);
}

int PassiveTCP(char *service, int qlen) {
    return passivesock(service, "tcp", qlen);
}

/*	A worker thread that handles a client on the specified socket	*/
void *ExecuteTCPWorker(void *args) {
	char *token;
	int ssock, alen, sendLen, found;
	char TCPData[ARRAYLEN], sendData[ARRAYLEN], receivedData[ARRAYLEN],
combo[IPADDRLEN];

	ssock = (int)args;
    printf("Entered TCP Worker for Error Collection Server .... %d\n\n",
ssock);

	while(TRUE) {
	    alen = ReadLine(ssock, receivedData, sizeof receivedData);
		strcpy(TCPData, receivedData);

		/*	The user has sent a request to quit	*/
		if(strcmp(TCPData, "QUIT:BYE") == 0) {
			printf("\nError Collection Server with id: %d has
				QUIT.\n\n", ssock);
			fflush(stdout);
			break;
		}
		else {
			printf("RECEIVED the following Routers from Error
				Collection Server: %d\n", ssock);
			printf("%s\n\n", TCPData);

			sendData[0] = '\0';
			token = strtok(TCPData, ",");

			/*	Search for the existance of any of the routers in the
				linked list	*/
```

```
                        found = FALSE;
                        while (token != NULL) {
                               /* Process the token */
                               if (searchNodes(routers, token) == FALSE) {
                                      strcat(sendData, token);
                                      strcat(sendData, ",");
                               }
                               else
                                      found = TRUE;

                               /* Get the next word */
                               token = strtok(NULL, ",");
                        }

                        /*     Found duplicates    */
                        if (found == TRUE) {
                               sendLen = strlen(sendData);
                               sendData[sendLen-1] = '\0';
                               WriteLine(ssock, sendData, strlen(sendData));
                        }
                        else {
                               strcpy(TCPData, receivedData);
                               token = strtok(TCPData, ",");
                               while (token != NULL) {
                                      /* Process the token */
                                      sprintf(combo, "%d", ssock);
                                      strcat(combo, ":");
                                      strcat(combo, token);

                                      addNode(&routers, combo);

                                      /* Get the next word */
                                      token = strtok(NULL, ",");
                               }

                               strcpy(sendData, "NODUPLICATES");
                               WriteLine(ssock, sendData, strlen(sendData));
                        }

                        printf("\nSENDING:%s\n\n", sendData);
                        fflush(stdout);
                }
        }

    pthread_exit(0);
}

/*     Display the list of routers maintained in the Linked list   */
void DisplayRouterList() {
       displayNodes(routers);
}

void LoadArrayFromFile() {
       int addrLen;
       FILE *configFile;
       char ipAddr[IPADDRLEN];

    if(!(configFile = fopen("ASConfig.txt", "rb")))
        errexit("Cannot open the config file for the Address Server.\n");

       noOfRouters = 0;
       while(fscanf(configFile, "%s", ipAddr) > 0) {
              addNode(&routers, ipAddr);
```

```
	}

	fclose(configFile);
}

/*	The main thread that listens for clients	*/
void WaitForTCPClient() {
    int ssock, alen;
	int i, strMessageLen;
	char strMessage[120];
	pthread_t TCP_Thread;
    struct sockaddr_in fsin;

	while(TRUE)  {
		// Accept new client.
		alen = sizeof(fsin);
		ssock = accept(msock, (struct sockaddr *)&fsin, &alen);

		sprintf(strMessage, "Error Collection Server with id: %d and IP:
			%s connected on port: %s ...",
				ssock, inet_ntoa(fsin.sin_addr), service);
		strMessageLen = strlen(strMessage);

		printf("%s\n", strMessage);
		for (i = 0; i < strMessageLen; i++) printf("*");
		printf("\n\n");

		if(ssock < 0)
			errexit("accept failed: %s\n", sys_errlist[errno]);

		// Start a new thread to process requests from this new client.
		pthread_create(&TCP_Thread, NULL, ExecuteTCPWorker,
			(void *)ssock);
		sleep(1);
	}
}

/*	An interrupt handler routine that gets called when the user presses
	CTRL+C */
void catch_int(int sig_num) {
	int Choice;

	printf("\b\b");
	printf("Continue               ... 1\n");
	printf("Re-load the file       ... 2\n");
	printf("List ECS & its Routers ... 3\n");
	printf("Exit to prompt         ... 4\n\n");
	printf("Enter your choice: ");
	while(Choice != 1 && Choice != 2 && Choice != 3 && Choice != 4) {
		scanf("%d", &Choice);
	}

	printf("\n");
	if(Choice == 1) {
		signal(SIGINT, catch_int);
		WaitForTCPClient();
	}
	else if(Choice == 2) {
		signal(SIGINT, catch_int);
		LoadArrayFromFile();
		DisplayRouterList();
		WaitForTCPClient();
	}
```

```
       else if(Choice == 3) {
              signal(SIGINT, catch_int);
              DisplayRouterList();
              WaitForTCPClient();
       }
       else if(Choice == 4) {
              printf("\nTerminating the Server....\n");
              close(msock);
              printf("\nGood-Bye...\n\n");

              exit(0);
       }
}

int main(int argc, char *argv[]) {
    service = argv[1];

       routers = NULL;
       signal(SIGINT, catch_int);
    msock = PassiveTCP(service, QLEN);
       LoadArrayFromFile();

       printf("Server started on service: %s\n\n", service);
    WaitForTCPClient();

    return(0);
}
```

```
/************************************************************************
*      ErrorExit.c                                                      *
*      Defines a routine to display error messages and quit             *
************************************************************************/


#include <varargs.h>
#include <stdio.h>

/*------------------------------------------------------*
 *     errexit - print an error message and exit                        *
 *------------------------------------------------------*/

int errexit(char *format, ...) {
       va_list args;

       va_start(args);
       _doprnt(format, args, stderr);
       va_end(args);
       exit(1);
}


/************************************************************************
*      PassiveSock.c                                                    *
*      Defines a routine that initializes and listens to a TCP          *
*      socket on a specific port                                        *
************************************************************************/

#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h>
#include <netdb.h>

#ifndef INADDR_NONE
    #define INADDR_NONE    0xffffffff
#endif

extern int    errno;
extern char   *sys_errlist[];

u_short portbase = 0;

/*     Create a TCP socket and bind it to the specified port */
int passivesock(char *service, char *protocol, int qlen) {
       struct servent *pse;
       struct protoent *ppe;
       struct sockaddr_in sin;
       int sock, type;

       bzero((char *)&sin, sizeof(sin));
       sin.sin_family = AF_INET;
       sin.sin_addr.s_addr = INADDR_ANY;

       if(pse = getservbyname(service, protocol))
              sin.sin_port = htons(ntohs((u_short)pse->s_port)+portbase);
       else if((sin.sin_port = htons((u_short)atoi(service))) == 0)
                     errexit("can't get \"%s\" service entry\n", service);

       if((ppe = getprotobyname(protocol)) == 0)
              errexit("can't get \"%s\" protocol entry\n", protocol);
```

```
	if(strcmp(protocol, "udp") == 0)
		type = SOCK_DGRAM;
	else
		type = SOCK_STREAM;

	sock = socket(PF_INET, type, ppe->p_proto);
	if(sock < 0)
		errexit("can't create socket: %s\n", sys_errlist[errno]);

	if(bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0)
		errexit("can't bind to %s: %s\n", service, sys_errlist[errno]);

	if(type == SOCK_STREAM && listen(sock, qlen) < 0)
		errexit("can't listen on %s port: %s\n", service,
			sys_errlist[errno]);

	return sock;
}


/*************************************************************
*	LinkedList.c                                         *
*	Defines a set of routines to create and manipulate a linked *
*	list                                                 *
*************************************************************/

#include <stdlib.h>
#include <string.h>

#ifndef FALSE
	#define FALSE 0
#endif
#ifndef TRUE
	#define TRUE  1
#endif

struct node {
	char data[20];
	struct node *link;
};

void addNode(struct node **q, char data[20]) {
	struct node *temp, *r;

	if(*q == NULL) {
		temp = malloc(sizeof(struct node));
		strcpy(temp->data, data);
		temp->link = NULL;
		*q = temp;
	}
	else {
		temp = *q;

		/* go to the last node */
		while(temp->link != NULL)
			temp = temp->link;

		/* add the node to the end */
		r = malloc(sizeof(struct node));
		strcpy(r->data, data);
		r->link = NULL;
		temp->link = r;
	}
```

```
}

void removeNode(struct node **q, char data[20]) {
	char *position;
	struct node *old, *temp;
	int dataLen, foundPosition;

	dataLen = strlen(data);
	temp = *q;
	while(temp != NULL) {
		position = strstr(temp->data, data);
		if (position == NULL) {
			old = temp;
			temp = temp->link;
			continue;
		}

		foundPosition = temp->data - position;
		if (foundPosition < 0)
			foundPosition *= -1;

		if(foundPosition == 0) {
			/* check if the node to be deleted is the first node */
			if(temp == *q) {
				*q = temp->link;
				/* free the memory occupied by the node */
				free(temp);
				return;
			}
			else {
				old->link = temp->link;
				free(temp);
				return;
			}
		}
		else {
			old = temp;
			temp = temp->link;
		}
	}
}

int searchNodes(struct node *q, char data[20]) {
	int i, j, found;
	char server[20], router[20];

	found = FALSE;
	while(q != NULL) {
		for(i = 0; q->data[i] != ':'; i++);
		for (i = i+1, j = 0; i < strlen(q->data); i++)
			router[j++] = q->data[i];
		router[j] = '\0';

		if (strcmp(router, data) == 0) {
			found = TRUE;
			break;
		}

		q = q->link;
	}

	return found;
}
```

```
void displayNodes(struct node *q) {
	int i, j;
	char server[20], router[20];

	printf("\nECS\t\tROUTER\n", server, router);

	while(q != NULL) {
		for(i = 0; q->data[i] != ':'; i++)
			server[i] = q->data[i];
		server[i] = '\0';

		for (i = i+1, j = 0; i < strlen(q->data); i++)
			router[j++] = q->data[i];
		router[j] = '\0';

		printf("%-4s\t\t%-4s\n", server, router);
		q = q->link;
	}
	printf("\n");
	fflush(stdout);
}

int countNodes(struct node *q) {
	int c;

	c = 0;
	while(q != NULL) {
		q = q->link;
		c++;
	}

	return c;
}



/*****************************************************************
*	ASConfig.txt                                                 *
*	Defines a Error Collection Server, router pair               *
*	used to populate the Address Server with initial dummy data *
*****************************************************************/

10:1
10:2
```

```
/*****************************************************************
*      ErrorCollectionServer.c                                   *
*      This program is a Client & talks to the Address Server    *
*      using Socket on TCP port.                                 *
*****************************************************************/

#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/ipc.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <netdb.h>

#define TCP_PORT    9887

extern int   errno;
extern char  *sys_errlist[];

#define IPADDRLEN   20
#define MAXROUTERS  10
#define ARRAYLEN    IPADDRLEN * MAXROUTERS + 10

#define      FALSE        0
#define TRUE        1

#define SEND 1
#define      QUIT   2

#ifndef INADDR_NONE
    #define INADDR_NONE    0xffffffff
#endif

int noOfRouters, maxRouterValue;
char routers[MAXROUTERS][IPADDRLEN], newRouters[MAXROUTERS][IPADDRLEN];

void PrintMenu() {
       printf("\n1....Send Routers.\n");
       printf("2....Quit.\n");
       printf("Enter your choice: ");
}

int InitializeTCP(char *ServAddr) {
       int sockID;
       struct hostent * hostentry;
       struct sockaddr_in serv_addr;

       printf("%s\n", ServAddr);
       bzero((char *) &serv_addr, sizeof(serv_addr));

       hostentry = gethostbyname(ServAddr);
       if ((hostentry == NULL) || (hostentry->h_addr_list == NULL)) {
              printf("Client: gethostbyname returned an error.\n");
              exit(1);
       }

       memcpy(&(serv_addr.sin_addr), hostentry->h_addr_list[0],
              hostentry->h_length);
       serv_addr.sin_family             = AF_INET;
       serv_addr.sin_port         = htons(TCP_PORT);
```

```
	if((sockID = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
		printf("Client: cannot open stream socket.\n");
		exit(1);
	}

	if(connect(sockID, (struct sockaddr *)&serv_addr,
					sizeof(serv_addr)) < 0) {
		printf("Client: can't connect to the server.\n");
		exit(1);
	}

	return sockID;
}

/*	Writes N bytes to the output stream specified  */
int WriteN(register int fd, register char *ptr, register int nbytes) {
	int	nleft, nwritten;

	nleft = nbytes;
	while (nleft > 0) {
		nwritten = write(fd, ptr, nleft);
		if (nwritten <= 0)
			return(nwritten);		/* error */

		nleft -= nwritten;
		ptr   += nwritten;
	}

	return(nbytes - nleft);
}

/*	Writes N bytes to the output stream specified along with a new line
character	*/
int WriteLine(int fd, char *ptr, int nbytes) {
	ptr += nbytes;

	if (*(ptr-1) != '\n') {
		*ptr = '\n';
		ptr -= nbytes++;
	}

	return WriteN(fd, ptr, nbytes);
}

/*	Read a line from the specified stream	*/
int ReadLine(register int fd, register char *ptr, register int maxlen) {
	int	n, rc;
	char	c;

	for (n = 1; n < maxlen; n++) {
		if ( (rc = read(fd, &c, 1)) == 1) {
			if (c == '\n')
				break;
			*ptr++ = c;
		} else if (rc == 0) {
			if (n == 1)
				return(0);	/* EOF, no data read */
			else
				break;		/* EOF, some data was read */
		} else
			return(-1);		/* error */
	}
```

```
	*ptr = 0;
	return(n);
}

void ClearBuffer(char *buffer) {
	int i;

	for(i = 0; i < ARRAYLEN; i++) buffer[i] = '\0';
}

void LoadArrayFromFile() {
	int addrLen;
	FILE *configFile;
	char ipAddr[IPADDRLEN];

    if(!(configFile = fopen("ECSConfig.txt", "rb")))
	 errexit("Cannot open the config file for the Error Collection
		Server.\n");

	noOfRouters = 0;
	while(fscanf(configFile, "%s", ipAddr) > 0) {
		strcpy(routers[noOfRouters++], ipAddr);
	}
	maxRouterValue = atoi(routers[noOfRouters-1]);

	fclose(configFile);
}

void PrintRouters() {
	int i;

	for(i = 0; i < noOfRouters-1; i++) {
		printf("%s, ", routers[i]);
	}
	printf("%s\n\n", routers[i]);
}

int main(int argc, char *argv[]) {
	int sockID, op, i, j, k, found, countDuplicates, difference;
	char sendRouters[ARRAYLEN], receivedRouters[ARRAYLEN];
	char tempRouters[ARRAYLEN], *token;

	if(argc < 2) errexit("usage: ErrorCollectionServer.out IP Address.\n");

	sockID = InitializeTCP(argv[1]);
	/*	Populate the array of routers from the configuration file	*/
	LoadArrayFromFile();
	printf("Searched and found the following routers: ");
	PrintRouters();

	/*	Continue search for routers until user quits	*/
	while(TRUE) {
		ClearBuffer(sendRouters);
		ClearBuffer(receivedRouters);

		PrintMenu();
		scanf("%d", &op);
		printf("\n");

		if(op == SEND) {
			printf("Sending the following routers to the Address
				Server.\n");
```

```
sendRouters[0] = '\0';
for(i = 0; i < noOfRouters-1; i++) {
       strcat(sendRouters, routers[i]);
       strcat(sendRouters, ",");
}
strcat(sendRouters, routers[i]);
printf("%s\n\n", sendRouters);
fflush(stdout);

/*     Continue search for routers until all routers are not
       unique */
while(TRUE) {
       WriteLine(sockID, sendRouters, strlen(sendRouters));
       ReadLine(sockID, receivedRouters,
                     sizeof receivedRouters);

       /*     An error occured on the server for the list of
              routers sent  */
       if(strcmp(receivedRouters, "ERROR") == 0) {
              printf("The routers recieved on the Address
                      Server were not valid.\n\n");
              fflush(stdout);
              break;
       }
       /*     The list of routers sent to the server are
              unique */
       else if(strcmp(receivedRouters, "NODUPLICATES") == 0)
       {
              printf("\n\nNone of the routers that was send
                     are being served by other Error
                     Collection Servers");
              printf("\n\nThe following routers can be
                       served by this Error Collection
                       Server\n%s\n\n", sendRouters);
              fflush(stdout);
              break;
       }
       /*     Not all routers in the list sent were unique
              hence search for more      */
       else {
              printf("The following routers are already
                       being served by other Error Collection
                       Servers\n");
              strcpy(tempRouters, receivedRouters);

              countDuplicates = j = 0;
              token = strtok(tempRouters, ",");
              while (token != NULL) {
                     /* Process the token */
                     strcpy(newRouters[j++], token);

                     /* Get the next word */
                     token = strtok(NULL, ",");
              }

              /*     Create a new list to be sent to the
                     server */
              for(i = 0; i < noOfRouters; i++) {
                     found = FALSE;
                     for(k = 0; k < j; k++) {
                            if(strcmp(newRouters[k],
                                    routers[i]) == 0) {
                                   found = TRUE;
```

```
                                                break;
                                        }
                                }
                                if(found == FALSE) {
                                        if(countDuplicates == 0)
                                                printf("%s", routers[i]);
                                        else
                                                printf(",%s", routers[i]);
                                        countDuplicates++;
                                }
                        }

                        difference = countDuplicates;

                        j = 0;
                        token = strtok(receivedRouters, ",");
                        while (token != NULL) {
                                strcpy(routers[j++], token);
                                token = strtok(NULL, ",");
                        }

                        /*      Simulate the searching of routers */
                        printf("\n\nSearching for more routers\n\n");
                        for (i = maxRouterValue+1;
                                i < maxRouterValue+difference+1; i++) {
                                sprintf(routers[j++], "%d", i);
                        }
                        maxRouterValue += difference;

                        sleep(3);
                        printf("Found and Sending the following
                                 routers to the Address Server.\n");

                        sendRouters[0] = '\0';
                        for(i = 0; i < noOfRouters-1; i++) {
                                strcat(sendRouters, routers[i]);
                                strcat(sendRouters, ",");
                        }
                        strcat(sendRouters, routers[i]);
                        printf("%s\n\n", sendRouters);
                        fflush(stdout);
                }
        }
        /*      User has requested a quit  */
        else if(op == QUIT) {
                sprintf(sendRouters, "QUIT:BYE");
                WriteLine(sockID, sendRouters, strlen(sendRouters));
                break;
        }
        else printf("Not a valid choice...Try Again...\n");
        fflush(stdout);
}

close(sockID);
return 0;
}
```

```
/*****************************************************************
*      ErrorExit.c                                                *
*      Defines a routine to display error messages and quit       *
*****************************************************************/


#include <varargs.h>
#include <stdio.h>

/*-------------------------------------------------------*
 *     errexit - print an error message and exit         *
 *-------------------------------------------------------*/

int errexit(char *format, ...) {
       va_list args;

       va_start(args);
       _doprnt(format, args, stderr);
       va_end(args);
       exit(1);
}


/*****************************************************************
*      ECSConfig.txt                                              *
*      Defines a list of routers for the Error Collection Server  *
*      used to populate it with initial dummy data                *
*****************************************************************/

1
2
3
4
5
```

## APPENDIX B

*Client:* A client is the requesting program or user in a client/server relationship. For example, the user of a Web browser is effectively making client requests for pages from servers all over the Web. The browser itself is a client in its relationship with the computer that is getting and returning the requested HTML file. The computer handling the request and sending back the HTML file is a server. [8]

*Server:* a server is a computer program that provides services to other computer programs on the same or other computers. In the client/server programming model, a server is a program that awaits and fulfills requests from client programs in the same or other computers. [8]

*Link:* a link is a physical (and, in some usages, a logical) connection between two points.

*Network:* A network is a series of points or nodes interconnected by communication paths. Networks can interconnect with other networks and contain sub networks. The most common topology or general configurations of networks include the bus, star, and token ring topologies. Networks can also be characterized in terms of spatial distance as local area networks (LAN), metropolitan area networks (MAN), and wide area networks (WAN). [8]

*Internet 2:* Internet 2 is collaboration among more than 100 U.S. universities to develop networking and advanced applications for learning and research. Since much teaching,

learning, and collaborative research may require real-time multimedia and high-bandwidth interconnection, a major aspect of Internet 2 is adding sufficient network infrastructure to support such applications. [8]

*Bridge:* a bridge is a product that connects a local area network (LAN) to another local area network that uses the same protocol (for example, Ethernet or token ring). You can envision a bridge as being a device that decides whether a message from you to someone else is going to the local area network in your building or to someone on the local area network in the building across the street. A bridge examines each message on a LAN, "passing" those known to be within the same LAN, and forwarding those known to be on the other interconnected LAN (or LANs). [8]

*Router:* a router is a device or, in some cases, software in a computer, that determines the next network point to which a packet should be forwarded toward its destination. The router is connected to at least two networks and decides which way to send each information packet based on its current understanding of the state of the networks it is connected to. A router is located at any gateway (where one network meets another), including each Internet point-of-presence. [8]

*LAN:* A local area network (LAN) is a group of computers and associated devices that share a common communications line and typically share the resources of a single processor or server within a small geographic area (for example, within an office building). Usually, the server has applications and data storage that are shared in common by multiple

computer users. A local area network may serve as few as two or three users (for example, in a home network) or many as thousands of users (for example, in an FDDI network). [8]

*WAN:* A wide area network (WAN) is a geographically dispersed telecommunications network. The term distinguishes a broader telecommunication structure from a local area network (LAN). A wide area network may be privately owned or rented, but the term usually connotes the inclusion of public (shared user) networks. An intermediate form of network in terms of geography is a metropolitan area network (MAN). [8]

*MAN:* A MAN (metropolitan area network) is a network that interconnects users with computer resources in a geographic area or region larger than that covered by even a large local area network (LAN) but smaller than the area covered by a wide area network (WAN). The term is applied to the interconnection of networks in a city into a single larger network (which may then also offer efficient connection to a wide area network). It is also used to mean the interconnection of several local area networks by bridging them with backbone lines. The latter usage is also sometimes referred to as a campus network. [8]

*Real Time:* A process is considered to be in real time if it guarantees a certain capability within a specified time constraint. Real time describes a human rather than a machine sense of time. [8]

*Analysis tool:* A piece of software that retrieves the source of problems all along the network by contacting the “address server” which stores the addresses of “error collection servers” that maintain error logs.

*Error Collection Server:* A set of mini computers that are placed all across the network to monitor and maintain log of potential problems on the network. These servers send probe packets to the network at regular intervals to check the network status.

*Address Server:* A single mini computer that has a well known address, and which maintains a list of all active error collection servers on the network. It provides an easier means of contacting error collection servers, as it reduces the need for the knowledge of the addresses of all error collection servers. It is the address server that can be used to draw a virtual map of the entire network from the information it maintains.

*Packet:* A packet is the unit of data that is routed between an origin and a destination on the Internet or any other packet-switched network. When any information is sent from one place to another on the Internet, the Transmission Control Protocol (TCP) layer of TCP/IP divides the file into "chunks" of an efficient size for routing. Each of these packets is separately numbered and includes the Internet address of the destination. The individual packets for a given file may travel different routes through the Internet. When they have all arrived, they are reassembled into the original file (by the TCP layer at the receiving end). [8]

*ICMP:* Internet Control Message Protocol is a message control and error-reporting protocol between a host server and a gateway to the Internet. ICMP uses Internet Protocol (IP) datagram, but the messages are processed by the IP software and are not directly apparent to the application user. [6, 8]

*IP:* The Internet Protocol (IP) is the method or protocol by which data is sent from one computer to another on the Internet. Each computer (known as a host) on the Internet has at least one IP address that uniquely identifies it from all other computers on the Internet. When you send or receive data (for example, an e-mail note or a Web page), the message gets divided into little chunks called packets. Each of these packets contains both the sender's Internet address and the receiver's address. [8]

*TCP:* Transmission Control Protocol is a set of rules (protocol) used along with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. While IP takes care of handling the actual delivery of the data, TCP takes care of keeping track of the individual units of data (called packets) that a message is divided into for efficient routing through the Internet. [8]

*Gateway:* A gateway is a network point that acts as an entrance to another network. On the Internet, a node or stopping point can be either a gateway node or a host (end-point) node. Both the computers of Internet users and the computers that serve pages to users are host nodes. The computers that control traffic within your company's network or at your local Internet service provider (ISP) are gateway nodes. [8]

## REFERENCES

[1] Ted Hanss “Internet2 End-to-End Performance Initiative or Why Fat Pipes Aren't Enough” TERENA Networking Conference, Antalya, Turkey (May 2001)

[2] Bandula W. Abeysundara and Ahmed E. Kamal “High-speed local area networks and their performance” ACM Computing Surveys CSUR (Jun. 1991): 221 – 264.

[3] Thomas W. Doeppner, Philip N. Klien and Andrew koyfman “Using Router Stamping to Identify the Source of IP Packets” Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS 2000) Athens Greece (Nov. 2000): 184-189.

[4] Gorry Fairhurst. (2001). Introduction to routers.
[On-line] http://www.erg.abdn.ac.uk/users/gorry/course/inet-pages/router.html

[5] Terry Gray. (2001). “Finger Pointing Tools” for Isolating Distributed System Performance Problems
[On-line] http://staff.washington.edu/gray/papers/fpt.html

[6] J. Postel. (1981). Internet Control Message Protocol, DARPA Internet Program Protocol Specification
[On-line] http://www.ietf.org/rfc/rfc0792.txt

[7] Denial-of-Service Attack via ping
[On-line] http://ciac.llnl.gov/ciac/bulletins/h-18.shtml

[8] Network hardware related terms
[On-line] http://whatis.techtarget.com/definitionsCategory/0,289915,sid9_tax1681,00.html

[9] What is Jini?
[On-line] http://di002.edv.uniovi.es/~falvarez/whatisjini.pdf