12-14-2001

Implementation and Analysis of the IP Measurement Protocol (IPMP)

Steven Michael Carter

IMPLEMENTATION AND ANALYSIS OF THE IP MEASUREMENT PROTOCOL

(IPMP)

By

Steven M. Carter

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science

Mississippi State, Mississippi

December 2001

IMPLEMENTATION AND ANALYSIS OF THE IP MEASUREMENT PROTOCOL

(IPMP)

By

Steven M. Carter

Approved:

Donna S. Reese Associate Professor of Computer Science (Director of Thesis)	Rainey Little Associate Professor of Computer Science (Committee Member)
Anthony Skjellum Associate Professor of Computer Science (Committee Member)	Julian E. Boggess Associate Professor of Computer Science (Graduate Coordinator)
A. Wayne Bennett Dean of the College of Engineering	

Name: Steven Michael Carter

Date of Degree: December 14, 2001

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Donna Reese

Title of Study: IMPLEMENTATION AND ANALYSIS OF THE IP MEASUREMENT PROTOCOL (IPMP)

Pages in Study: 55

Candidate for Degree of Master of Science

The increased size and complexity of the Internet necessitates a more substantial measurement protocol than is currently available. This work explores the IP Measurement Protocol, providing background information, covering the development of a reference implementation, and finally comparing its accuracy, overhead, and ease of implementation to the current generation of protocols used in network measurement.

Vmware, a hardware simulation application, was used to simulate a network on which to test IPMP, as well as compare it to current generation tools. Ipmp_ping, a tool written to test IPMP, was pitted against ping and traceroute in order to attain round trip time, one-way delay, and path discovery measurements. The accuracy and overhead of these tools were compared to each other.

Although ipmp_ping had more overhead than ping when measuring round trip time, it was just as accurate and more capable. Ipmp_ping proved to be much more efficient than traceroute with similar accuracy. Overall, ipmp_ping was as accurate and had negligibly more or significantly less overhead than the tools it was compared to while providing more functionality and being easy to implement.

ACKNOWLEDGEMENTS

The author expresses his special thanks to his major advisor, Dr. Donna Reese, for her invaluable guidelines and support throughout the entire period of this research. Also, thanks are due to Tony McGregor and the National Laboratory for Applied Network Research for proposing and facilitating this work.

TABLE OF CONTENTS

Page

CHAPTER Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

In the past five years, the Internet has seen an explosive growth in population, size, and complexity. This growth has been due in part to the increased amount of commodity traffic. The Internet is becoming a more integral part of the world's daily business. Because of the Internet's growing importance, there is a need to keep it running at peak performance with little interruption in service.

Coupled with the Internet's increased usage is an increase in its size and complexity. With multiple paths between countless entities and continents, the Internet's complexity has increased several orders of magnitude since its inception and has even expanded beyond the ability of network engineers to fully understand it. Even the individuals who participate in the designing and building of the individual carriers' networks do not have a complete understanding of the Internet's overall structure.

The current generation of Internet Protocols lacks the ability to provide statistics needed to compile the desired metrics. Many of these tools either use ICMP (Internet Control Message Protocol), some proprietary protocol with limited scope, or a protocol carried over TCP or UDP. Although these tools get their job done with a reasonable amount of success, they are based on protocols that are fundamentally unsuited for the job of network measurement.

Internet Control Message Protocol, or ICMP, was originally designed as a protocol to send control messages across the network (Comer 1998). The most widely used functionality of ICMP is the ability to send echo requests and receive responses, such as in "ping." Most network measurement utilities are based on this ability or some derivative of it. Since network measurement was more a peripheral issue in ICMP, the protocol lacks the needed mechanisms to provide effective network measurement.

ICMP has also received much negative publicity due to its use as a mechanism for conducting denial of service attacks. This bad press has caused several ISPs (Internet Service Providers), network carriers, and end points to either limit or deny ICMP through their networks. Being treated differently from other traffic severely limits the effectiveness that ICMP has as a network monitoring protocol.

It is the hypothesis of this thesis that network measurement protocols designed with proper facilities to gather needed network metrics will provide more accurate and more efficient methods for monitoring and analyzing the Internet and next generation internets.

CHAPTER II

LITERATURE REVIEW

The increasing size and complexity of the Internet coupled with its ever-increasing importance poses serious problems in maintaining the stability and performance of the Internet. In recent years, several projects have been undertaken to deal with the problem of monitoring an extremely large network like the Internet. These projects include the Surveyor project, the National Internet Measurement Infrastructure project (Adams 1998), and the National Laboratory for Applied Network Research's Network Analysis Infrastructure project (McGregor 2000).

Although each project has subtle differences in the exact scope and execution of their measurements, they all share the common goal of an infrastructure of measurement equipment and software used to gather statistics about the overall network condition. Each of these projects uses a system of network probes distributed throughout the Internet or particular regional or experimental network of interest. These probes are used to collect and store data on the metrics of interest.

Most of the measurement projects separate their structure and method for administration from the actual tools used to collect data on the particular metric of interest. Some of these projects use existing tools and some use tools that were

developed within the project. This separation between structure and collection method makes it easy for these projects to adopt other tools as they are developed. Although the tools are based on the current generation of protocols, tools based on new protocols can easily be adopted.

2.1 Surveyor

Surveyor is one project with multiple worldwide participants for the purpose of network measurement. "[Surveyor is] based on standards work being done in the IETF's IPPM WG, (IP Performance Metrics Working Group). Surveyor measures the performance of the Internet paths among participating organizations" (Kalidindi 1999).

Surveyor is one project in which new protocols have been developed to accomplish the measurement goals. One such protocol is the One-Way Delay and Packet loss protocol or OWDP. OWDP measures one-way delay and packet loss on a link using UDP as its transport. OWDP takes a generic approach to the problem by developing a means for getting at a particular metric instead of merely developing a tool that stretches current protocols. Unfortunately, OWDP only measures one-way delay and packet loss and is insufficient to address the overall problem of network monitoring.

2.2 National Internet Measurement Infrastructure (NIMI)

Another project, the National Internet Measurement Infrastructure (NIMI), uses tools based on current protocols for the purpose of network measurement. Initial funding

for NIMI came from the National Science Foundation. NIMI is currently funded by DARPA to measure the global Internet. NIMI is based on Vern Paxson's Network Probe Daemon and was designed to be scalable and dynamic. “NIMI is scalable in that NIMI probes can be delegated to administration managers for configuration information and measurement coordination. It is dynamic in that the measurement tools are external to nimid as third party packages that can be added as needed" (NIMI 2000). Two of these tools are Poip and TReno.

Poip (Poisson Ping) is a tool designed to measure one-way delay and packet loss characteristics of a particular path (Paxson 1998). Not actually a form of ping, Poip is another protocol carried over UDP. It works by sending and receiving UDP packets at poisson intervals. It uses a generic "wire time" library and includes several sanity and packet integrity tests. As shown later in this paper, Poip is an example of a tool that uses an excessive amount of overhead to conducts its measurements. Although UDP adds less overhead because of its stateless, unreliable nature, it still adds a non-trivial amount that can be avoided.

"TReno is designed to measure the single stream bulk transfer capacity over an Internet path. It is a combination of two existing algorithms: traceroute and an idealized version of the flow control algorithms present in Reno TCP” (Mathis 2000). TReno uses either ICMP ECHO or low ttl UDP packets to solicit ICMP errors. Each ICMP error contains the sequence number of the packet that caused the response. Although Treno’s method relies on standard portions of the IP and ICMP protocols, it uses several protocols to accomplish its measurements. It uses ICMP for some aspects and UDP packets for

others. Although ICMP was originally designed to be used in this manner, the ttl functionality of IP was not. The ttl of a packet was originally meant to be used as a way to protect against routing loops that cause a packet to loop infinitely. In addition to creating a considerable amount of overhead, generating packets in such a way as to intentionally exceed the ttl is contrary to its original intent and can have side effects such as needlessly incrementing counters leading to a false indication of routing loops.

2.3 National Analysis Infrastructure (NAI)

The National Laboratory for Applied Network Research's (NLANR) contribution to the network measurement infrastructure effort is the National Analysis Infrastructure (NAI). NAI is composed of several components: active measurement, passive measurement, SNMP (Simple Network Measurement Protocol) and BGP (Border Gateway Protocol) data from participating routers and servers. The passive measurement project consists of several packet "sniffers" that derive workload and other traffic information from packet header traces. Like the other projects, the Active Measurement Program (AMP) consists of approximately 100 probes that measure round trip times, packet loss, and topology.

2.4 IP Performance Metrics Working Group

To support the work of monitoring the Internet, the Internet Engineering Task Force formed an IP performance metrics working group (IPPM WG). The purpose of this working group is to provide a framework in which the various measurement projects

can better accomplish their goals. "The IPPM WG will define specific metrics, cultivate technology for the accurate measurement and documentation of these metrics, and promote the sharing of effective tools and procedures for measuring these metrics. It will also offer a forum for sharing information about the implementation and application of these metrics, but actual implementations and applications are understood to be beyond the scope of this working group" (IPPM WG 1999).

The IPPM working group is working in a top-down manner, creating guidelines from a generic point-of-view and converging on more specific criteria. For example, the first IPPM RFC (Request for Comments) starts by defining the notion of metrics and measurements. The IPPM working group further defines methods for collecting data, clock accuracy issues, the concept of "wire-time", and specific network metrics. Currently, the IPPM working group has defined metrics for connectivity, one-way delay, one-way packet loss, and round trip delay.

By using the IPPM definitions and guidelines, the network measurement community can better define the scope and compare the data from the various measurement efforts. Many of the existing projects, including NLANR's NAI , use the IPPM framework in development by the IETF.

2.5 IP Measurement Protocol

IPMP is a proposed Internet standard to be used for measurement of the modern Internet. The primary reason for IPMP's existence is to answer the question, "Where are the network delays occurring?" "[IPMP] operates as an echo protocol allowing packet

loss, path length, RTT and in some cases, one-way delay measurement" (McGregor 1998). IPMP was designed to be easy to implement, efficient, and used between IPMP un-aware devices.

IPMP has the following goals and features, including but not limited to:

- **Measurement of protocol based priority queuing**

The IPMP head includes a queue type field to specify how the echoing system and the intermediate routers schedule the packet if they implement a packet scheduling discipline that is not FIFO. If the queue type is specified and a non-FIFO discipline is used on the router, the IPMP packet must be scheduled as if it were a packet from the IP protocol specified in Queue Type. For example, a Queue Type of 6 means schedule the packet as if it were a TCP packet (McGregor 1998). In this way, an IPMP packet can measure more specifically the delays inherent in the way that a particular protocol's packets are processed in the individual routers.

- **Support for forward and reverse path measurements of a single packet**

The IPMP packet includes provisions for the optional inclusion of a path record by each router the packet passes though. In addition to the address of the router, a path record includes a timestamp. The addition of the path record will obviously change the checksum of the packet. The checksum can be re-computed in either an absolute or a relative manner. Although more complicated, re-computing the checksum in a relative manner can help to lessen the effect of IPMP on busy routers. Since the path record is optional, it can be left out if the router does not support the functionality or is too busy to expend the cycles.

- **Supports bit error rate measurements**

IPMP provides an indication of how often a packet needs to be transmitted due to an error in the transmission of a single bit.

- **Allows accurate RTT measurements**

IPMP has mechanisms to account for clock skew between two end stations. These mechanisms allow IPMP to collect more accurate measurements even when the clock at the network monitoring station and the clock at the echoing station are out of sync.

- **Reduces the measurement overhead on the network**

One of the primary goals of IPMP is simplicity. Because it was designed with network measurement in mind, there is no need to use higher level protocols like UDP or TCP. In the case of TCP, extra overhead is incurred due to connection setup, flow control, and other factors associated with a reliable connection protocol. Although UDP doesn't have the same kind of overhead that TCP does, there are extra headers associated with it as well. In addition, both TCP and UDP require more processing on both the monitoring and echoing nodes as well as on potentially each router in between.

IPMP also saves overhead by incorporating needed functions such as path records. Whereas tools like traceroute use errors caused by sending multiple UDP packets with varying time-to-live values and relying on error reporting to discover the path, IPMP can accomplish the same function with a single packet.

Another benefit of IPMP lies in its extremely low overhead. Since it requires about as much processing overhead as IP forwarding, the opportunity for it to be used as

a means for a denial-of-service attack is diminished. This makes it more likely that IPMP will be treated no differently from other traffic.

2.6 Luckie IPMP Implementation

In the Spring of 2000, after the beginning of this work, Matthew Luckie implemented IPMP in Tony McGregor's Advanced Communications and Network Systems class at Waikato University in New Zealand. The implementation includes IPMP echo request and echo response functionality. In addition, Luckie developed a driver program with which to test his implementation.

The document that accompanied the implementation contained some good background information on how the FreeBSD networking stack worked, confirming the research done in this thesis. Unfortunately, there was very little analysis of IPMP itself and how it compared to current day protocols.

The conclusion of the document is that IPMP "answers the requirement for a protocol that allows Internet measurement teams to measure networks more richly and accurately" (Luckie 2000). As is shown later in this document, this is not entirely true.

2.7 Summary

There are several tools currently being used for network measurement. Some of these tools use existing protocols and some of the tools use new protocols. The tools using existing protocols either are not capable of getting the desired metrics or use the existing protocols in an unintended manner, causing inefficiency and even inaccuracy in

the measurements. Unfortunately, even though there are new protocols for network measurement, they are limited in scope and unable to gather a reasonable range of measurements.

For this reason, a protocol such as IPMP is needed in order to satisfy the needs of the network measurement community. In order for this to happen, an implementation of IPMP needs to be tested against existing tools and protocols to verify its efficiency, accuracy, and flexibility.

CHAPTER III

WORK INVOLVED

3.1 Implementation

The implementation phase consisted of the coding and testing of IPMP as well as any changes to the protocol specification that are necessitated by issues found while coding. The implementation phase ended with the testing of the implementation.

3.2 Testing and Comparison

The second part of this work is the testing and comparison of the IPMP implementation. One obvious problem with testing is that the protocol would not be needed if there were a way to gather the network characteristics needed to check IPMP. There are no existing active network measurement protocols that can authoritatively provide the needed truth to compare IPMP results against.

One method of testing is to build a network with known characteristics on which to run the IPMP protocol. Since the attributes of the network are known, the characteristics of interest can be calculated and used to compare against the experimental data. This method is not without problems, however. One problem lies in the inability to build a test network with all of the intricacies of a large network such as the Internet.

There will inevitably be issues that are encountered when IPMP is deployed on a real network with real problems.

One possible solution to this is to use passive monitors. Passive monitors, although more difficult to implement and maintain, can gather the needed statistics to compare with the IPMP data. Since passive monitors can be deployed on a real network, it is more likely that problems directly related to measuring a large network will be pointed out. One difficulty in implementing this testing method is the overhead in placing passive monitors throughout the Internet.

Yet another possibility is the utilization of a software-simulated network. Getting comparable measurements would either require an expansive network to create large enough wire times or very precise measurement probes to be able to determine the smallest differences in time. Both of these methods are quite expensive in addition to the basic requirements of at least two routers and three hosts. Software simulated networks allow for the building and instrumentation of a test network on a single host. The simulated network should provide for the accurate comparison of IPMP to existing protocols without requiring a costly test network.

3.3 Completion Standards

It is important to define the scope of this project since it is hoped that its life will progress far beyond this thesis project. The goal of the implementation phase is to incorporate all features mentioned in the IPMP draft proposal (McGregor 1998). The

only features and fixes incorporated that are not in the IPMP draft document will be due to issues uncovered due to the project itself.

Since IPMP is merely a protocol that will be used as a facility for collecting network measurement and not an actual tool, creating a driver program for the testing phase would be a project in itself. Therefore, each and every feature will not be tested since the overhead involved in such testing is high and goes beyond the scope of this project. The goal of the testing phase is to test those features most used in network measurement tools today, such as connectivity, round-trip time, and path.

CHAPTER IV

EXPERIMENTAL SETUP

4.1 Operating System

In choosing development platforms, several factors had to be considered. First and foremost, the platform had to allow kernel development since the implementation of IPMP could not be accomplished in user accessible code. This limited the choice of operating systems to those that were based on UNIX. This is not necessarily because non-UNIX operating systems do not allow kernel access, but because they provide the easiest and most cost affective access in terms of software needed to develop kernel level code. Non-UNIX operating systems such as the Windows variants do allow the addition of kernel level hooks, but all of the tools from the compilers to the operating system itself are required to be purchased. In addition, since the kernel code is proprietary, the kernel source is not available to explore, hindering the developmental process. For these reasons, a UNIX operating system seems the best choice for the development of this project.

There are several UNIX-based and UNIX-derived operating systems from which to pick. Linux, FreeBSD, and Solaris are all supported by Vmware and would each be possible candidates for the project. Both Linux and FreeBSD are free and their kernel source code is readily available. However, it is widely accepted that FreeBSD has a more

robust and better thought out networking stack. This coupled with the fact that FreeBSD is widely used in the development of networking protocols make FreeBSD the best candidate.

FreeBSD makes an excellent development choice for several reasons. First and foremost it and all required development tools are free, and its source is readily available. FreeBSD distributions come with a full complement of development tools and network monitoring programs. It is also easy to install. Finally, the FreeBSD kernel is well constructed and documented and its kernel facilities make it easy to develop kernel level code.

4.2 Compiler

The choice of compilers was rather simple. The GNU C compiler is a product of the GNU project originally started at MIT. It is freely available and widely distributed. The FreeBSD distribution comes with the GNU C compiler installed and the kernel source uses it by default for compilation.

4.3 Test Environment

There exist several problems in testing a protocol developed for network testing. Since the premise of IPMP is that there is no good protocol for taking network measurements, the only way to test a network measurement protocol is to use numerous network probes, a network with known attributes, or a software simulated network. Both the use of network probes and the construction of a network with known characteristics

require the utilization of several host and networking components. Also, the number of hosts and networking components increases due to the fact that the protocol must be tested in a routed environment. This requires a minimum of three hosts and equipment required for the construction of two separate networks. For this reason, a network simulated in software was used to build the test environment.

Vmware is a product that allows one or more virtual machines to be run on a host machine. Along with these virtual machines, Vmware supplies virtual networks to connect the virtual machines. The advantage of using Vmware is clear; there is no need to construct a physical network of physical machines. Since Vmware simulates hardware, it can be used to obtain an accurate testing environment that mimics a real one.

Using Vmware provides other benefits. It is expensive to deploy enough probes to accurately monitor the testing of IPMP. Since the entire testing environment runs on the same host machine, they all use the same clock. This means that each of the virtual machines should have the same rate and drift characteristics. Using a time synchronization protocol such as ntp, the virtual machine's clocks should be able to be kept reasonably close enough to take proper measurements. This allows for the use of simple tools, such as tcpdump, instead of expensive network probes.

The test environment, shown in Figure 4.1, consists of three virtual machines (freebsd0, freebsd1, freebsd2) and three virtual networks (network0, network1, network2). One virtual network, network0, connects all of the virtual machines to the host machine in order to share files and control the test virtual machines. Virtual network1 connects virtual machines freebsd0 and freebsd1. Virtual network2 connects

virtual machines freebsd1 and freebsd2. Each of the test virtual machines has a route to the host machine. Freebsd0 and freebsd2 use freebsd1 as their default gateway. In this way, the only path from freebsd0 to freebsd2 is through freebsd1. This setup provides a simple routed network in which to test IPMP.

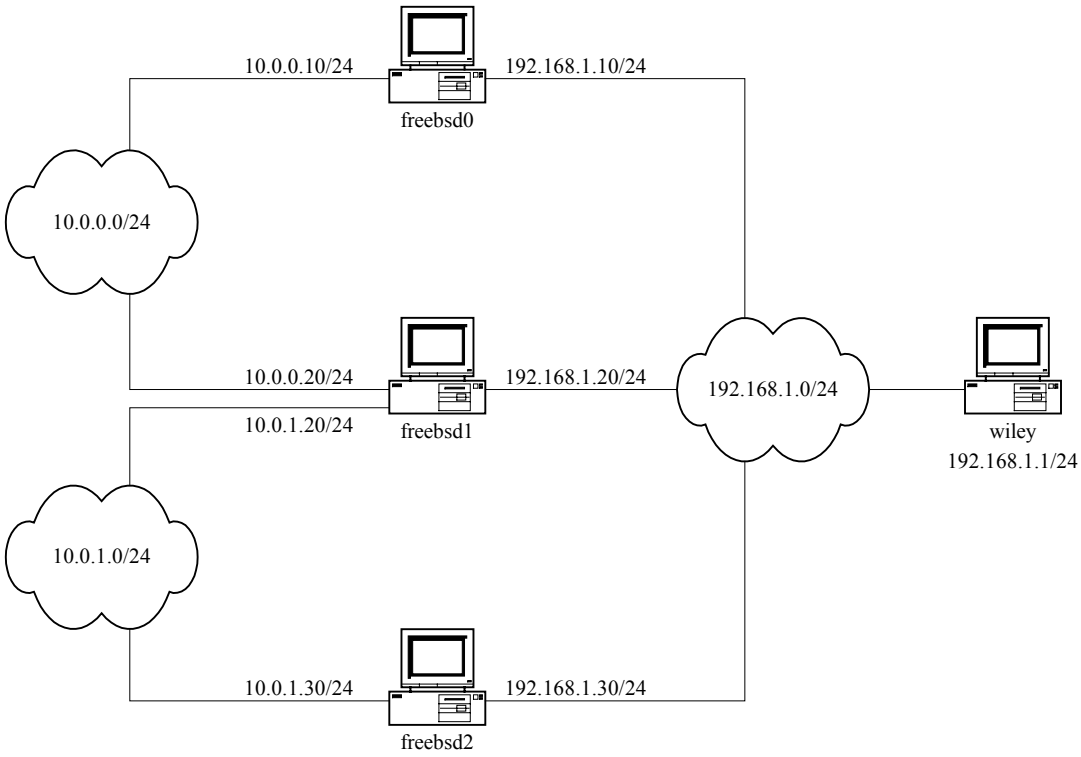

Figure 4.1 Test network diagram

CHAPTER V

CODE DEVELOPMENT

5.1 Background

To add another protocol to the FreeBSD kernel, an overall understanding of the packet flow through the kernel is essential. As shown in Figure 5.1, the flow begins with the receipt of the packet from the physical media. The packet is first passed to ip_input as an mbuf. An mbuf, or memory buffer, is a construct that allows for the manipulating of the memory containing the packet. There are several tools provided to allocate, concatenate, free, and convert mbufs. Using mbufs allows for the efficient manipulating of the packet data to achieve the best performance with the least effort.

In ip_input, a check is made to decide whether the packet is for the local machine or some other machine. If the packet is to be delivered locally, the protocol number is used in order to find the correct handling routines out of the protocol switch structure. From the switch structure, the packet could be dispatched to one of several protocol handlers including TCP, UDP, ICMP, and IPMP. The particular protocol handler is passed the mbuf to be processed. In case of a protocol such as IPMP, the processed mbuf can be handed to ip_output to be sent as a response to the sending machine.

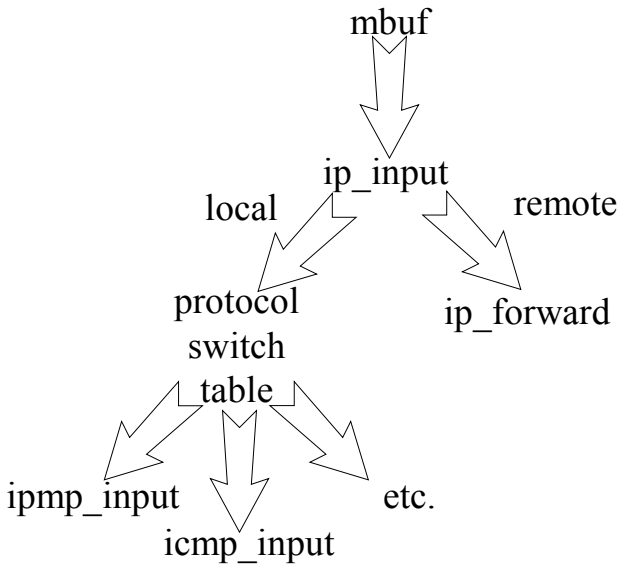

Figure 5.1 Packet flow through FreeBSD kernel

If it is decided that the packet is destined to another machine, the packet is passed to ip_forward for any necessary processing before it is sent. From here, protocols such as IPMP can usurp the packet in order to augment its contents.

5.2 Foundation

The IPMP code is modeled on the existing ICMP code in the FreeBSD kernel. The reason for this approach is simple; the authors of the existing code most likely have more experience than I, and deriving the IPMP from the ICMP would provide a good foundation. Although ICMP is considerably more complicated than IPMP, it provided a good foundation because it provides similar functionality to IPMP. In addition, both ICMP and IPMP are layer 3 protocols. Complexity aside, ICMP interacts with the

networking stack in all of the same places that IPMP will need to interact. So, in addition to providing a good foundation for IPMP, using the ICMP code also provides a trail to follow through the kernel to find all the relevant places in the kernel that IPMP hooks needed to be added.

5.3 Kernel Hooks

There are two basic functions that IPMP has to be involved in: handling packets destined for the host and routings packets destined for other hosts. The modifications start with inserting the hooks into the FreeBSD IP layer code to support IPMP. This is where using ICMP as a model was particularly useful. By searching through the existing files for tags relevant to ICMP, a map was made of places where hooks into the network code needed to be placed for IPMP. Since ICMP is used to accomplish more tasks than IPMP is intended to provide, it was not necessary to add code for every occurrence of an ICMP function in the network stack. Hooks were only added in those places that IPMP's functionality warranted.

Since IPMP is a layer 3 protocol, the first step was to allocate a protocol number. Originally, a random free protocol number was chosen. However, in order to test with Luckie's implementation, protocol number 169 was used. The protocol is assigned in the system include file /usr/src/sys/netinet/in.h as shown in Figure 5.2.

```
/*
 * Added by for IPMP support
 */
#define IPPROTO_IPMP            169
```

Figure 5.2 Addition of IPMP protocol number

The protocol number is used to select the module to which the IP packet will the sent. If the protocol number is IPPROTO_TCP, the packet will be delivered to the TCP module in the transport layer. Similarly, if the protocol number is IPPROTO_IPMP, the packet will be given to the IPMP module.

FreeBSD uses a protocol switch structure to decide which of the various interface functions it needs to call for the various protocols. The type definition for the protocol switch structure is found in /usr/include/sys/protosw.h, and is shown in Figure 5.3.

Making the needed substitutions for the protocol number and the input function to the protocol and leaving the other values as set by ICMP, we get the inetsw definition in /usr/src/sys/netinet/in_proto.c as shown in Figure 5.4.

This structure acts as a dispatch table, matching protocol numbers to the input function of the appropriate module. In addition to the protocol switch structure, sysctl support for IPMP can be added in this file. Sysctl is a utility to get and set kernel state. It is useful for changing kernel level values from user land code.

The last hook to add, shown in Figure 5.5, was in /usr/src/sys/netinet/ip_input.c. This file contains the ip_forward function that FreeBSD uses to forward packets from one interface to another when being used in a router capacity. Although the packet is not

```
struct protosw {
short   pr_type;            /* socket type used for */
struct  domain *pr_domain;  /* domain protocol a member of */
short   pr_protocol;        /* protocol number */
short   pr_flags;           /* see below */
/* protocol-protocol hooks */
void    (*pr_input) __P((struct mbuf *, int len));
                            /* input to protocol (from below) */
int     (*pr_output)      __P((struct mbuf *m, struct socket *so));
                            /* output to protocol (from above) */
void    (*pr_ctlinput)__P((int, struct sockaddr *, void *));
                            /* control input (from below) */
int     (*pr_ctloutput)__P((struct socket *, struct sockopt *));
                            /* control output (from above) */
/* user-protocol hook */
void    *pr_ousrreq;
/* utility hooks */
void    (*pr_init) __P((void)); /* initialization hook */
void    (*pr_fasttimo) __P((void));
                          /* fast timeout (200ms) */
void    (*pr_slowtimo) __P((void));
                          /* slow timeout (500ms) */
void    (*pr_drain) __P((void));
                          /* flush any excess space possible */
struct  pr_usrreqs *pr_usrreqs; /* supersedes pr_usrreq() */
};
```

Figure 5.3 Protocol switch table definition

```
struct protosw inetsw[] = {
[…]
{ SOCK_RAW,     &inetdomain, IPPROTO_IPMP, PR_ATOMIC|PR_ADDR,
  ipmp_input,   0,              0,              rip_ctloutput,
  0,
  0,            0,              0,              0,
  &rip_usrreqs
},
[…]
};
```

Figure 5.4 IPMP addition to protocol switch table

destined to the host, a path record must be appended to the packet traversing the host. In ip_forward, the packet is temporarily usurped and passed to the ipmp_append_pathrecord function to have a path record appended to it.

```
if(ip->ip_p == IPPROTO_IPMP)
{
    ipmp_append_pathrecord(m);
}
```

Figure 5.5 Addition of IPMP path record in ip_forward

Adding the appropriate code in the places shown enabled the networking code to call the appropriate IPMP function whether the packet is destined for the host or traversing it. With the hooks put into the networking code, the next step was to change the ICMP code to make it handle IPMP.

5.4 First Revision

As mentioned before, the IPMP code development started with ICMP as a template. The ICMP source files are kept in the netinet directory of the FreeBSD kernel source tree. Ip_icmp.c includes all of the functions relevant to the operation of ICMP. These functions include: icmp_error, icmp_input, icmp_reflect, icmp_send, ip_next_mtu, and badport_bandlim. It is intuitively obvious to all but the most casual observer that only a subset of these functions needs to be mirrored into IPMP.

The include file ip_icmp.h contains all generic ICMP definitions and icmp_var.h contains all definitions pertaining to the particular implementation of ICMP in FreeBSD. This file structure was utilized to form the basis of IPMP.

The first revision did nothing more than use the copied structure of ICMP to send echo send and receive packets with no changes other than the protocol number. Success

at this point gave confirmation of a successful base on which to build. It verified that all of the appropriate places in the networking stack were located and modified in order to support IPMP.

The next step involved striping down the myriad of unneeded functions in the adopted ICMP code and changing the packet header to the definition in the IPMP draft specification.

When a packet is passed to icmp_input from the protocol switch table, the checksum is verified and the message type is inspected. Operations are performed specific to the packet type. If the packet must be re-sent after processing, it is given to the icmp_reflect function, which among other things, switches the source and destination addresses and re-sets the ttl. From the icmp_reflect function, the packet is passed to icmp_send. Icmp_send calculates the checksum then passes it back to the network layer to be sent on to its destination. This structure is warranted given ICMP's complexity. However, ICMP's complex structure is not needed for the simple tasks that IPMP has to carry out. For this reason, much of this complexity was stripped out of later versions of the IPMP code.

5.5 IPMP File Structure

As with ICMP, the main body of the IPMP codes resides in ip_ipmp.c. Unlike ICMP, it is composed of only two main functions: ipmp_input and ipmp_append_pathrecord. These two functions, along with the API provided by the FreeBSD networking stack, is all that is needed to accomplish the implementation of

IPMP. Likewise, the include file structure also mirrors the include file structure of ICMP discussed earlier. Ip_ipmp.h contains all definitions needed for IPMP in general. Ipmp_var.h contains all definitions specific to this implementation of IPMP.

5.6 The Journey Begins

An IPMP packets journey begins with the ipmp_input function called from the networking stack's protocol switch.

```
void ipmp_input(register struct mbuf *ipmp_mbuf; int off, proto);
```

Figure 5.6 Ipmp_input prototype

After Ipmp_input is called with a pointer to an mbuf containing the packet, the header offset, and protocol (Figure 5.6), the first step is to check to see if the entire IPMP packet was received. This is accomplished using m_pullup as seen in Figure 5.7. M_pullup takes as an argument the mbuf pointer and a minimum buffer size. The fragments of the mbuf are coagulated into a contiguous piece of memory. If the fragments don't add up to at least the minimum packet size, the m_pullup command fails and the packet is discarded. The minimum packet size is calculated by taking the size of the ipmp header structure and adding the IP header. The size of the packet falls below this value, some part of the structure was lost and the packet needs to be discarded. The packet is discarded by freeing the mbuf and returning from the function call.

```
i = hlen + sizeof(struct ipmp);
if (ipmp_mbuf->m_len < i && (ipmp_mbuf = m_pullup(ipmp_mbuf, i)) == 0)
{
            /* icmpstat.icps_tooshort++; */
            if (ipmpprintfs) {
                  printf("IPMP packet too short.\n");
            }
goto freeit;
}
```

Figure 5.7 Checking of packet size with m_pullup

After a successful m_pullup call, all pointers into the mbuf must be reset since it is likely that the addresses into the packet have changed.

As delivered, the mbuf points to the beginning of the entire IP packet. To get access to the IPMP portion of the packet, we have to adjust the mbuf pointer by offsetting it the size of the IP header. Once this is done, a pointer is assigned to the beginning of the IPMP portion of the packet. With the mbuf pointer adjusted, the mbuf is passed to the checksum routine. If the checksum calculation is incorrect, then the mbuf is freed and the function returns. With the pointer assigned and the checksum calculated, the mbuf pointers are returned to their original positions. This process is shown in Figure 5.8.

```
ipmp_mbuf->m_len -= hlen;
ipmp_mbuf->m_data += hlen;
ipmp = mtod(ipmp_mbuf, struct ipmp *);
if (in_cksum(ipmp_mbuf, ipmplen)) {
  /* icmpstat.icps_checksum++; */
  goto freeit;
}
ipmp_mbuf->m_len += hlen;
ipmp_mbuf->m_data -= hlen;
```

Figure 5.8 Manipulation of mbuf to access IPMP packet and perform checksum

This ends the generic processing section of the ipmp_input function. The rest of the actions are applied based on the packet type.

5.7 Packet Type Processing

The meat of the ipmp_input function is a switch statement to perform different functions depending on the packet type. The four currently supported packet types are echo request, echo response, info request, and info response.

As seen in Figure 5.9, the first step in handling an IPMP Echo Request is to change the packet type and exchange the source and destination address. This readies the packet to be sent back and be recognized as an echo response. Then the returned ttl field is set to the ttl of the received ipmp echo request packet so that it is not overwritten with the new ttl value. In this way, the sending host knows the number of hops on both the forward and reverse paths. Knowing this information can help detect routers that are either not configured to insert IPMP path records or who are too busy to do so. The last step is to append the path record to the ipmp packet.

```
switch (ipmp->ipmp_type) {
case IPMP_ECHOREQUEST:
      ipmp->ipmp_type = IPMP_ECHOREPLY;
      tempaddr = ip->ip_dst;
      ip->ip_dst = ip->ip_src;
      ip->ip_src = tempaddr;
      ipmp->ipmp_returned_ttl = ip->ip_ttl;
      ipmp_append_pathrecord(ipmp_mbuf);
      break;
```

Figure 5.9 IPMP echo request processing

The next type of packet to handle is an IPMP echo response. An IPMP echo response is received when the host previously sent out an IPMP echo request. Since this is the end of the packet's life cycle, the only job that needs to be done here is to append the final path record as seen in Figure 5.10. The packet is then passed to the waiting ipmp_ping or other user land program that sent the packet.

```
case IPMP_ECHOREPLY:
      ipmp_append_pathrecord(ipmp_mbuf);
      break;
```

Figure 5.10 IPMP echo reply processing

For an IPMP info request packet, the first action performed is to change the packet type to IPMP info reply. Although the IPMP info request and the IPMP info reply headers are different, the first 8 bytes are constructed the same and therefore the response is able to use the same first 8 bytes as the IPMP info reply packet. An mbuf for the rest of the IPMP info request packet is allocated. The size of the new mbuf containing the additional section of the IPMP info reply is added to the length of the previous IPMP info request packet. The length, performance data pointer, accuracy, and processing overhead fields are then filled in. Currently, these fields are either not supported, or not relevant to this implementation. Finally, the two mbufs are concatenated together to make the entire IPMP info reply packet. The handling of the info request packet is shown in Figure 5.11.

```
case IPMP_INFOREQUEST:
      ipmp->ipmp_type = IPMP_INFOREPLY;


      inforeply_mbuf = m_get(M_DONTWAIT, MT_DATA);
      if(inforeply_mbuf == 0) return;
      inforeply_mbuf->m_len = sizeof(struct ipmp_info_reply);
      info_reply = mtod(inforeply_mbuf, struct ipmp_info_reply *);

      i = sizeof(struct ipmp_info_reply);
      ip->ip_len += i;
      info_reply->ipmp_length =
            htons(ntohs(info_reply->ipmp_length) + i);
      info_reply->ipmp_performance_data_pointer =
            info_reply->ipmp_length;
      info_reply->ipmp_ip_address  = ip->ip_src;
      info_reply->ipmp_accuracy  = 0;
      info_reply->ipmp_processing_overhead  = 0;

      m_cat(ipmp_mbuf,inforeply_mbuf);
      ipmp_mbuf->m_pkthdr.len += i;
      break;
```

Figure 5.11 IPMP info request processing

Like an IPMP echo response packet, this is the last leg of the journey for an IPMP information response packet. The only difference is that there need not be a path record appended to the end of the IPMP information response packet. The packet is simply handed to the awaiting user level application.

5.8 Appending the Path Record

The last of the IPMP code deals with the appending of a path record to an IPMP echo request or echo response involving the host. The ipmp_append_pathrecord is used both to append path records to packets sent from monitor hosts and reflected from echo hosts, as well as adding path records to packets that are passed through from one interface

to another. The appending of the path record was the only non-trivial function that needed to be carried out more than once. Therefore, it was broken out into its own function.

Since a pointer to the mbuf is passed in with the pointers pointing to the beginning of the entire IPMP packet, the pointers to the IPMP portion of the packet need to be re-assigned. This is the only duplicated code in the ipmp_append_pathrecord function.

Next, a mbuf is allocated to hold the path record structure and a pointer assigned to the beginning of the path record structure. The address of the host passing the packet, as well as a timestamp containing both whole seconds and fractional seconds is placed in the pathrecord structure. Depending on the performance implications, the timestamp can be either the true time, or some counter that is easily accessible from within the kernel of the routing device. The path pointer field in the IPMP header and the length field in the IP header are updated to reflect the new size of the packet. This process is shown in Figure 5.12.

The last step, shown in Figure 5.13, is to actually append the path record structure to the end of the packet, and calculate the checksum. The function then returns to either the ipmp_input function of the ip_forward function, depending on the stage of life the IPMP packet is in.

```
pathrecord_mbuf = m_get(M_DONTWAIT, MT_DATA);
if(pathrecord_mbuf == 0) return;
pathrecord_mbuf->m_len = sizeof(struct ipmp_pathrecord);

path              = mtod(pathrecord_mbuf, struct ipmp_pathrecord
*);
/* find the IP address to put in the path record */
IFP_TO_IA(ifp, ia);
if (ia)
  path->ip = IA_SIN(ia)->sin_addr;

getnanotime(&path->timestamp);
/* strncpy(ctime_buf, ctime(&path->timestamp.tv_sec), 24); */
ctime_buf[24] = '\0';
path->timestamp.tv_sec  = htonl(path->timestamp.tv_sec);
path->timestamp.tv_nsec = htonl(path->timestamp.tv_nsec);
```

Figure 5.12 Creation of IPMP path record

```
m_cat(ipmp_mbuf, pathrecord_mbuf);
ipmp_mbuf->m_pkthdr.len += i;

ipmp_mbuf->m_data += hlen;
ipmp_mbuf->m_len -= hlen;
ipmp->ipmp_checksum = 0;
ipmp->ipmp_checksum = in_cksum(ipmp_mbuf, ip->ip_len - hlen);
ipmp_mbuf->m_data -= hlen;
ipmp_mbuf->m_len += hlen;
ipmp_mbuf->m_pkthdr.rcvif = (struct ifnet *)0;
return;
```

Figure 5.13 Concatenation of IPMP path record

CHAPTER VI

RESULTS

IPMP will be compared to other protocols in two ways, accuracy and overhead. First, we will look at the overhead of IPMP for three metrics: one-way delay, round-trip time, and path discovery.

6.1 Vmware Shortcomings

Using Vmware was a benefit in the testing of the work, although not as much as was hoped. Vmware did provide a good environment in which to quickly assemble a network and test the functionality of IPMP. Building a comparable test network would have required considerably more time and resources and tied the experimentation to a particular facility. Using Vmware allowed for the use of a single, although well equipped, computer for the entire development cycle. Vmware also did a good job of simulating the hardware in a realistic, if not real-time, manner.

The original plan was to monitor the packet's travel by sniffing the virtual network interface. Unfortunately, the virtual network is implemented in a switched manner. Consequently, the network sniffing did not see packets that were not destined to or that did not originate from the interface being sniffed. As a result, sniffing the virtual

network as a whole could not be effective and an alternative was sought. In addition to the hindrance that a switched network presented to sniffing, there is also the fact that sniffing the network as a whole would still only provide an indication of when the packet was put onto the network, not when it was received by a particular interface. This problem could be overcome by outfitting the Vmware networking code itself to give an indication of when a particular network host "received" the packet off of the network. In addition to being more work that fell outside of the scope of the project, this method was rejected for reasons mentioned later in this chapter.

The next attempt was to sniff the ingress and egress interfaces for a particular network. In this manner, a timestamp of when the packet was "transmitted" and when the packet was "received" would be available. The differential in these two timestamps gives the "wire time" of the packets and therefore the desired measurement. Unfortunately, using probes placed on two different hosts presents a problem with keeping the two virtual machine's clocks synchronized. The ingress and egress interfaces on the virtual network exist on each of the three virtual machines. This means that the sniffer's timestamps are no longer relative to one clock, but to three. This means that any measurements taken are susceptible to the inaccuracies of the various clocks.

In order to attempt accurate measurements, the host's clocks have to be synchronized. Since all of the virtual hosts use the same hardware and therefore the same clock, they should be able to keep close enough to make accurate measurements.

For the first attempt to synchronize the clock, Network Time Protocol (NTP) was used. The NTP software on the host machine was configured to use its clock as its time

source since no outside clock was available. Each of the three virtual machines was told to use the host machine as its time server. Unfortunately, the clocks could not keep themselves close enough, often straying from the host machine's clock by 2-3 seconds. This variation fell outside of tolerances and caused the NTP sessions to be lost.

The degree in which the Vmware software simulates a machine was quite complete, simulating the hardware of three separate clocks.

The second attempt at measuring using the Vmware network was to use a GPS connected through the serial interface. Since the Vmware machines use the same hardware, they should be able to share the serial port. With the GPS hooked to the single shared serial interface, the Vmware machines should be able to share the GPS signal, enabling them to keep their clock synchronized. However, the three virtual machines were not able to share the same device. The first machine to get to the serial port would block access to the other two. The mechanisms available to put the serial port in a multi-access mode were unsuccessful. In addition, it is doubtful that even if each of the hosts was able to read the GPS signal, that the clock would be kept synchronized. This is due to the nature in which the NTP daemon works. NTP converges on the correct time using a number of samples. Because of the software simulation of the clock, it is likely that NTP would not be able to properly converge on the correct time.

Since all reasonable attempts to synchronize the clocks were exhausted, attempting to compare the protocol's numbers to real numbers were abandoned. In retrospect, these measurements would not have yielded extremely useful results due to the emulated nature of the test environment. Any results would have been skewed by the

manner in which Vmware needs to operate. For example, the Vmware processes being context switched through the processor in a quasi-random manner would render quasi-accurate results. In addition, the time differentials involved would have been so small that they would have been lost in the noise.

Where Vmware fell short was in its ability to provide a sufficient measurement environment. The very attribute that makes it desirable for the functional test, namely the utilization of a single computer for testing, contributed to its inability to be used to take measurements. Because of the manner in which the individual entities shared the hardware on a multi-user, non real-time, operating system, Vmware was unable to properly simulate the true nature in which an individual network entity operates separate from other networks entities. Sharing the same hardware meant that one node was able to affect another node in a way other than what it would on a network. This fact in itself makes Vmware unwise to use when taking measurements. Another problem is that the time a packet takes to traverse the virtual nodes and networks is so small that any differences between the respective measurements would be lost in the noise.

6.2 Round Trip Time

Ping is the most common network monitoring tool used today. Ping is the common name for the application that utilizes ICMP echo requests and echo responses to ascertain connectivity and round trip times. In order to function, a single ICMP echo request is sent to a host, which returns an ICMP echo request. The round trip time is

derived by simply taking the difference between the time the echo request is sent and the time the echo response is received.

On the test network, three ICMP packets were sent from freebsd2 to freebsd0 using ping. The output is shown in figure 6.1.

```
freebsd2# ping 10.0.0.10
PING 10.0.0.10 (10.0.0.10): 56 data bytes
64 bytes from 10.0.0.10: icmp_seq=0 ttl=254 time=2.409 ms
64 bytes from 10.0.0.10: icmp_seq=1 ttl=254 time=1.587 ms
64 bytes from 10.0.0.10: icmp_seq=2 ttl=254 time=1.472 ms
^C
--- 10.0.0.10 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.472/1.823/2.409/0.417 ms
```

Figure 6.1 Ping from freebsd2 to freebsd0

First, the packet arrives at freebsd1 since it is the gateway for the network on which freebsd0 resides. Although tcpdump sees that the packet is an ICMP echo request, there is no special processing required by the networking stack. Freebsd1 just forwards the echo request as it would any other IP packet as seen in Figure 6.2.

```
freebsd1# tcpdump -i lnc3
tcpdump: listening on lnc3
23:47:31.579947 freebsd2-net1 > freebsd0-net0: icmp: echo request
23:47:31.582834 freebsd0-net0 > freebsd2-net1: icmp: echo reply
23:47:32.584110 freebsd2-net1 > freebsd0-net0: icmp: echo request
23:47:32.585380 freebsd0-net0 > freebsd2-net1: icmp: echo reply
23:47:33.602035 freebsd2-net1 > freebsd0-net0: icmp: echo request
23:47:33.603328 freebsd0-net0 > freebsd2-net1: icmp: echo reply
```

Figure 6.2 Tcpdump output of freebsd1's network1 interface

Next, the echo request packet arrives at freebsd0 which formulates and returns an echo response packet as shown in Figure 6.3.

```
freebsd0# tcpdump -i lnc1
tcpdump: listening on lnc1
23:47:31.528005 freebsd2-net1 > freebsd0-net0: icmp: echo request
23:47:31.529728 freebsd0-net0 > freebsd2-net1: icmp: echo reply
23:47:32.526861 freebsd2-net1 > freebsd0-net0: icmp: echo request
23:47:32.528577 freebsd0-net0 > freebsd2-net1: icmp: echo reply
23:47:33.543177 freebsd2-net1 > freebsd0-net0: icmp: echo request
23:47:33.545025 freebsd0-net0 > freebsd2-net1: icmp: echo reply
```

Figure 6.3 Tcpdump of freebsd0's network0 interface

As the packets return to freebsd2, it calculates the return trip time to be 2.409ms, 1.587ms, and 1.472ms. Two possible reasons for the variance of the three times is the short path that the packet takes through the network and the non real-time nature of Vmware.

The header overhead involved in sending a pair of ICMP echo and response packets is relatively trivial. Each packet contains a 20 byte IP header, and a 8 byte, ICMP message giving a total header size of 28 bytes. An optional variable length data entry can be inserted into the echo request to help distinguish packet.

Comparing the header efficiency of ICMP to the header efficiency of IPMP becomes a bit more complex. Sending an IPMP echo request packet from freebsd2 to freebsd0 starts out with the IP header being the exact same size. Adding the 16 byte IPMP echo request header size to the 20 byte IP header size yields an overhead of 36 bytes. Like ICMP, IPMP can include variable length data as its payload. Where ICMP and IPMP differ in header size is with the first device that inserts a path record. An overhead of 12 bytes is incurred every time a path record is inserted. So for our test network, we have a total of 5 path records inserted. Two path records are added by

freedbsd2 itself, one before it enters the network, and one when it comes back. Freebsd1 also adds two path records since the packet passes through it on both the forward and reverse paths. And finally, freebsd0 adds a single path record midway through the journey. This gives us a total packet size of 96 bytes. This is more than 3 times greater than the header overhead for the ICMP echo request and echo response. In addition, the header overhead will increase as the number of hops in the path increases. It is obvious the overhead for IPMP is greater than the header overhead of ICMP when seeking round trip time.

6.3 One Way Delay and Path Discovery

Next we take a look at one way delay and path discovery. Traceroute is among the most popular ways of finding one-way delay measurements and path discovery. As with finding round trip time, finding one-way delay and path discovery starts with freebsd2. The traceroute output is shown in Figure 6.4.

```
freebsd2# traceroute 10.0.0.10
traceroute to 10.0.0.10 (10.0.0.10), 30 hops max, 40 byte packets
 1  freebsd1 (10.0.1.20)  3.188 ms  2.125 ms  1.034 ms
 2  freebsd0 (10.0.0.10)  5.185 ms  1.654 ms  2.328 ms
```

Figure 6.4 Traceroute from freebsd2 to freebsd0

As seen in Figure 6.5, traceroute sends three udp packets to freebsd1 each with a ttl of 1. As freebsd1 gets the packets, it decrements the ttl seeing that the packet must die. So it sends an ICMP time exceeded message back to freebsd2. In this way, freebsd2 is able to see what the next hop in the path is.

```
freebsd1# tcpdump -i lnc2
tcpdump: listening on lnc2
23:48:19.989212 freebsd2-net1.36421 > freebsd0-net0.33435: udp 12
[ttl 1]
23:48:19.989826 freebsd1-net1 > freebsd2-net1: icmp: time exceeded
in-transit
23:48:19.991151 freebsd2-net1.36421 > freebsd0-net0.33436: udp 12
[ttl 1]
23:48:19.991467 freebsd1-net1 > freebsd2-net1: icmp: time exceeded
in-transit
23:48:19.993108 freebsd2-net1.36421 > freebsd0-net0.33437: udp 12
[ttl 1]
23:48:19.993289 freebsd1-net1 > freebsd2-net1: icmp: time exceeded
in-transit
23:48:19.994294 freebsd2-net1.36421 > freebsd0-net0.33438: udp 12
23:48:19.998046 freebsd0-net0 > freebsd2-net1: icmp: freebsd0-net0
udp port 33438 unreachable
23:48:19.999312 freebsd2-net1.36421 > freebsd0-net0.33439: udp 12
23:48:20.001440 freebsd0-net0 > freebsd2-net1: icmp: freebsd0-net0
udp port 33439 unreachable
23:48:20.003276 freebsd2-net1.36421 > freebsd0-net0.33440: udp 12
23:48:20.005195 freebsd0-net0 > freebsd2-net1: icmp: freebsd0-net0
udp port 33440 unreachable
```

Figure 6.5 Traceroute traffic through freebsd1

With freebsd1 found, traceroute increments the ttl to 2 and sends three more packets. Three UDP packets arrive at freebsd0 with a ttl of 1. Like freebsd1, freebsd0 decrements the ttl by one and sees that the packet must die. Freebsd0 then sends an ICMP time exceeded message back to freebsd2, enabling freebsd2 to see freebsd0. Figure 6.6 shows the traffic flowing through freebsd0.

```
freebsd0# tcpdump -i lnc1
tcpdump: listening on lnc1
23:48:19.830295 freebsd2-net1.36421 > freebsd0-net0.33438:  udp 12
[ttl 1]
23:48:19.832281 freebsd0-net0 > freebsd2-net1: icmp: freebsd0-net0
udp port 33438 unreachable
23:48:19.832840 freebsd2-net1.36421 > freebsd0-net0.33439:  udp 12
[ttl 1]
23:48:19.834424 freebsd0-net0 > freebsd2-net1: icmp: freebsd0-net0
udp port 33439 unreachable
23:48:19.834755 freebsd2-net1.36421 > freebsd0-net0.33440:  udp 12
[ttl 1]
23:48:19.836101 freebsd0-net0 > freebsd2-net1: icmp: freebsd0-net0
udp port 33440 unreachable
```

Figure 6.6 Traceroute traffic through freebsd0

The overhead involved increases relatively dramatically when using traceroute. For each hop in the path, at least one UDP packet and one ICMP time exceeded packet needs to be sent. Each UDP packet includes at least a 20 byte IP header and a 12 byte UDP header, yielding a total of 32 bytes. Each ICMP time exceeded packet is also 32 bytes. So for each host, 64 bytes worth of packets need to be sent. For a network of two hosts, a minimum of 128 bytes is needed. Traceroute sends three packets for each hop for a total of 384 bytes. As hops are added, an additional 64 bytes per host is needed as compared to 24 bytes for IPMP. Although different implementations of traceroute may vary, IPMP has less overhead than traceroute for even the minimum case.

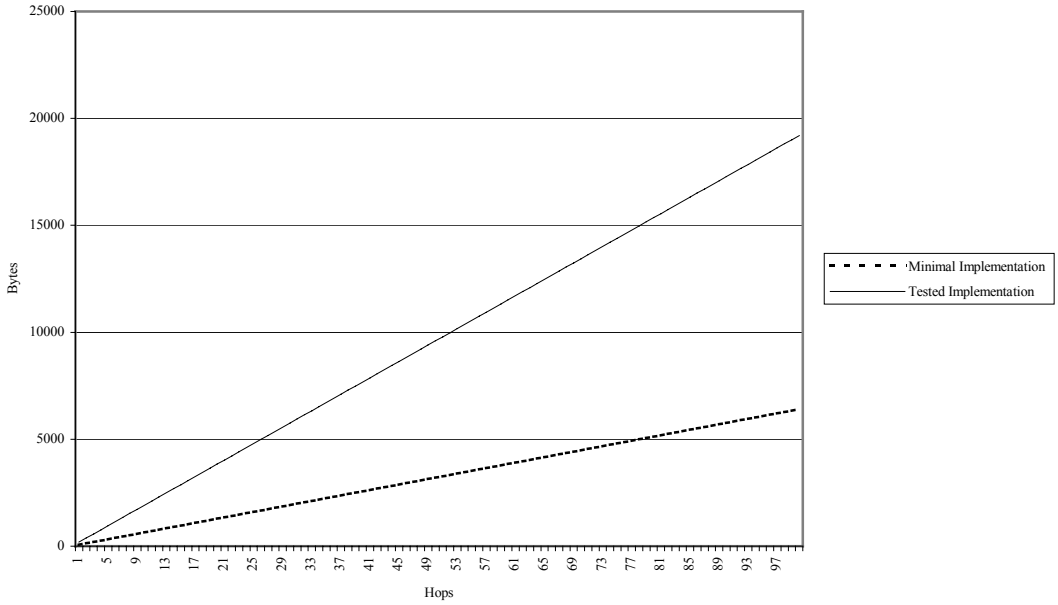

Figure 6.7 The increase in overhead using traceroute as the number of hops increases

As seen in Figure 6.7, the number of bytes used in route discovery by traceroute (solid line) increases rather dramatically as the number of hops goes up.

6.4 IPMP Ping

Now we take a look at using IPMP for attaining one-way delay, round trip time, and path discovery. Whereas previously, multiple commands were needed to attain these metrics, now we just need one. Ipmp_ping is a combination of user-level code and a kernel loadable module to send and receive IPMP echo request and echo reply packets. The kernel loadable modules are needed to formulate and send the IPMP echo request packets. From the output of this invocation of ipmp_ping, shown in Figure 6.8, we see that we get the 5 path records discussed earlier.

```
freebsd2# ./ipmp_ping 10.0.0.10
IP Measurement Protocol (IPMP) - Echo Test Routing
Sending the echo request...
Calling syscall 210 (ipmp_ping)
Waiting....
Parsing Echo Response...
IP   payload size 96
IPMP version      0
     queue type   6
     type         0
     ttl at turn  254
     return type  0
     length       60
    path pointer 96
    path records 5
 0        10.0.1.30 Fri Apr 20 23:46:53 2001  90103823
 1        10.0.1.20 Fri Apr 20 23:46:52 2001 947735249
 2        10.0.0.10 Fri Apr 20 23:46:52 2001 998242471
 3        10.0.0.20 Fri Apr 20 23:46:52 2001 982853025
 3  10.0.1.30 Fri Apr 20 23:46:53 2001 132477074
 4
```

Figure 6.8 Ipmp_ping from freebsd2 to freebsd0

In Figure 6.9, we see that the 16 byte IPMP echo request packet arrives at freebsd1 with a 12 byte path record inserted by freebsd2. Freebsd1 then inserts it's own path record and sends the packet on to freebsd0. On its return path, freebsd1 adds another path record and returns the packet to its origin.

```
freebsd1# tcpdump -i lnc3
tcpdump: listening on lnc3
23:46:52.950363 freebsd2-net1 > freebsd0-net0: ip-proto-169 28
23:46:52.986093 freebsd0-net0 > freebsd2-net1: ip-proto-169 64
```

Figure 6.9 Ipmp_ping traffic through freebsd1

Figure 6.10 shows the IPMP echo request packet arriving at freebsd0 with 2 path records. Freebsd0 adds its path record and sends the packet back to freebsd1.

```
freebsd0# tcpdump -i lnc1
tcpdump: listening on lnc1
23:46:52.999575 freebsd2-net1 > freebsd0-net0:  ip-proto-169 40
23:46:53.022557 freebsd0-net0 > freebsd2-net1:  ip-proto-169 52
```

Figure 6.10 Ipmp_ping traffic through freebsd0

The life cycle of the IPMP packets is relatively straightforward and efficient. With one packet, ipmp_ping is able to gather statistics on one-way delay, round-trip time, and path discovery.

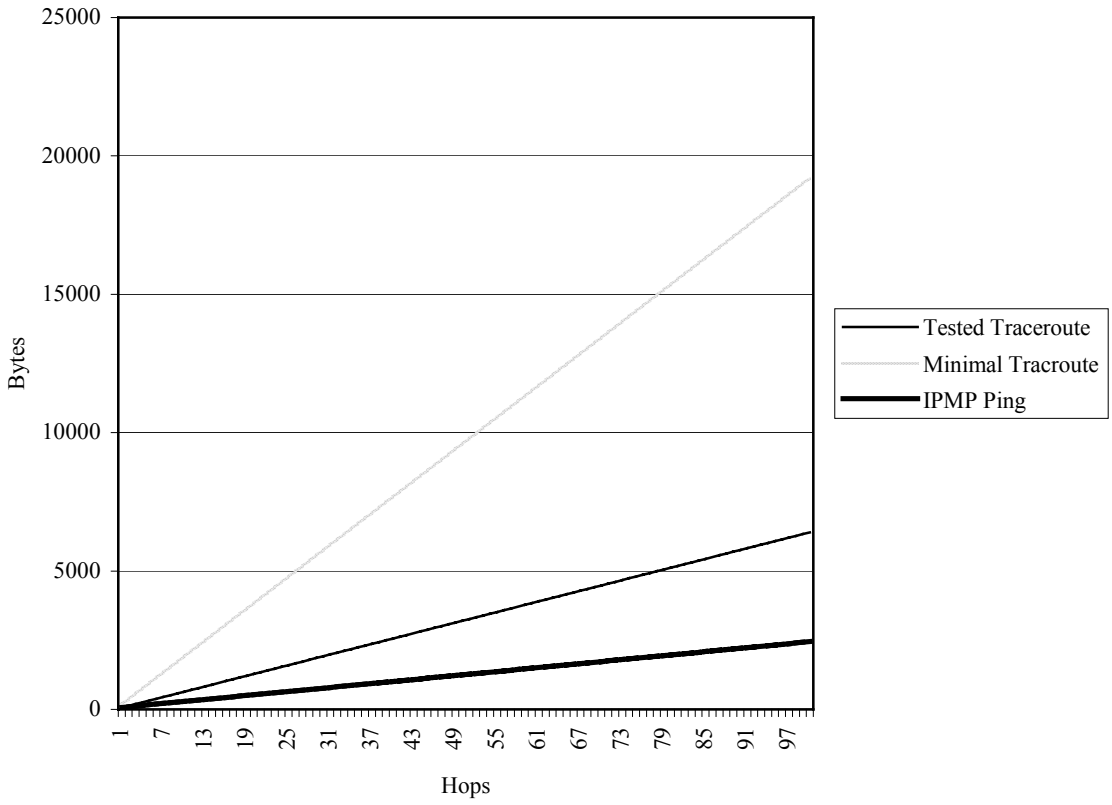

Figure 6.11 The increase of packet overhead of IPMP ping as the number of hops increases

In Figure 6.11, we can see the increase in the amount of traffic needed by IPMP as the number of hops increases. The graph shows that a little less than 2500 bytes are needed with 100 hops. This is less than even the minimal case of the traceroute method, which requires over 5000 bytes for the same number of hops.

6.5 Accuracy

Although it is difficult to test the accuracy of IPMP due to unsynchronized clocks, some level of verification can be reached. Looking at the tcpdump output from freebsd1 (Figure 6.8), a time difference of 35.7 ms exists between the passage of the IPMP packet on its forward and return paths. This agrees with the 35.1 ms time difference shown in the ipmp_ping output (Figure 6.7). Looking at the ping command and packet trace output, its accuracy was also verified in the same manner. In this way, the method used to verify IPMP's accuracy can be affirmed, increasing the confidence in the results. Although the numbers between ipmp_ping and ping are different, ipmp_ping is able to accurately take the required measurements.

6.6 Luckie Implementation

As noted earlier, another implementation of IPMP surfaced during the development of this implementation allowing for interoperability testing. The two implementations worked well with each other and yielded similar results. The only problem was the updating of the length field. The Luckie implementation increases the length field when adding path records. The specification says to leave the length field

unchanged when adding path records. This difference caused problems in determining the number of path records, but was overcome by changing ipmp_ping to use the path record pointer field in determining the number of path records.

The fact that the two implementations interoperated verifies that the IPMP draft was relatively straight-forward and easy to implement.

CHAPTER VII

CONCLUSIONS

7.1 Accuracy

Because of the problems with the testing environment, the results can not be used to proclaim absolute accuracy. However, there is sufficient data to prove that the measurements taken by IPMP are accurate within the constraints of the test environment.

7.2 Overhead

Although IPMP has more overhead than using simple pings, it provides much more information. In addition, IPMP proves to have much less overhead than traceroute while also providing more functionality. Therefore, IPMP does provide the statistics sought while using less packet overhead.

Another aspect of overhead, the impact on the router, is a cause for concern. Measurements show that the routers took an average of 8ms to process the IPMP packet. This is far greater than the 1ms average of a normal ping; however, the work required by an IPMP echo is obviously greater than the work required by an ICMP echo. This does not seem to be an unreasonable result, merely the price of added functionality. It should be noted that the Luckie implementation showed the packet processing times to be less

than 1ns. These results are suspect since the average PC's clock cycle is far greater than 1ns; therefore, the difference in two subsequent measurements must be greater than 1ns. In either case, IPMP allows for this by specifying an IPMP processing overhead field to provide measurement programs with the difference in processing time between an IPMP packet and a similar IP packet giving the ability to derive an accurate measurement.

However, even though the processing overhead does not affect the measurement, it does affect the router in a non-trivial manner. Today's core Internet routers handle a large and ever increasing amount of packets. They are able to do this in large part because of fast path routing done with application specific integrated circuits or ASICs. ASICs can handle the routing of a packet much more rapidly and efficiently than the router's management or general-purpose processors. The last thing that a network administrator wants is for a packet not to go through a router's fast path, because it can affect its performance greatly. Unfortunately, implementing IPMP in hardware would likely be a difficult proposition and only helpful for new routers. Likewise, special purpose network processors, if available in a particular router, are usually there to do another high impact function and would not be able to spare the processing cycles to handle IPMP. Therefore, IPMP packets would have to be handled by the router's management or general-purpose processors, which would be an unacceptable burden in many cases. In addition, the high processing overhead would leave the routers open to denial of service attacks by making the router process a large number of IPMP packets, causing it to get too busy to do its normal job. It is true that the router could choose to

eliminate much of the processing burden by choosing not to append a path record, but this would also eliminate much of the usefulness of IPMP.

7.3 Ease of Implementation

One of the benefits of IPMP is that it is easy to implement. Obviously, difficulty in implementing the protocol would mean that it is less likely to be implemented and therefore, IPMP would not be a viable protocol no matter how accurate and efficient it is. Keeping this fact in mind, ease of implementation is a rather important metric in the evaluation of IPMP.

IPMP only has two sets of functionality: echo request and echo response, info request and info response. Because of its lean nature, IPMP requires little code within the kernel of the hosts and routers. The majority of the functionality is accomplished by either moving a packet somewhere, or adding a pathrecord to a packet. The only calculation involved is in generating the checksum. However, even calculating the checksum is not difficult since most environments will provide this functionality. The most difficult part of implementing IPMP in FreeBSD was learning the FreeBSD kernel and how it handles packets. This learning curve would not be an issue for a company that is already familiar with the product that it produces.

The majority of the complexity involved in taking measurements with IPMP is in the driver program itself. The protocol allows for the measurements to be taken, but it is the driver program that is left to interpret the results and correlate and information request responses. However, the driver program only needs to be written once and ported to

different platforms. Therefore, even if the driver program were to figure into the implementation costs of IPMP, it would be a one time cost and worth the effort.

7.4 Overall Conclusion

The IPMP specification was well thought out and well designed. It achieved many of its goals, including accuracy, ease of implementation, and low packet overhead. Even though this overhead is relatively small, many of today's routers have very few processing cycles to spare. Therefore, it is the opinion of this work that it would be difficult to implement into any large production network in the foreseeable future.

REFERENCES

Adams, A., J. Mahdavi and M. Mathis. 1998. Architecture for large-scale internet measurement. *IEEE Communications Magazine*. 36: 48-54.

Comer, Douglas. 1988. *Internetworking with TCP/IP.* Englewood Cliffs, New Jersey: Prentice-Hall.

IPPM WG. 1999. *IP Performance Metrics (ippm).* http://www.ietf.org/html.charters/ippm-charter.html (Accessed 26 January 2000).

Kalidindi, S. and M. Zeauskas. 1999. *Surveyor: An infrastructure for the Internet Performance measurement.* http://telesto.advanced.org/~kalidindi/papers/INET/inet99.html (Accessed 26 January 2000).

Luckie, M. 2000. *Implementation of IPMP under FreeBSD.* A paper presented in Tony McGregor's Advanced Communications and Network Systems class.

Mathis, M. and J. Mahdavi. *Diagnosing internet congestion with a transport layer performance tool.* http://www.psc.edu/~mathis/htmlpapers/inet96.treno.html (Accessed 14 February 2000).

McGregor, A., H. Braun and J. Brown. 2000. *The NLANR network analysis infrastructure*. An article accepted for publication in IEEE Communications Magazine.

McGregor, A. 1998. *IP measurement protocol.* http://watt.nlanr.net/AMP/IPMP/ipmp.html (Accessed 26 January 2000).

NIMI. *Creating a National Internet Measurement Infrastructure.* http://www.ncne.nlanr.net/nimi/ (Accessed 25 January 2000).

Paxson, V., G. Almes, J. Mahdavi and M. Mathis. 1998. *Framework for IP performance metrics.* http://www.advanced.org/IPPM/docs/rfc2330.txt (Accessed 25 January 2000).

APPENDIX A

GLOSSARY OF TERMS

ICMP (Internet Control Message Protocol) - A protocol used for sending control messaged between networked hosts for such applications as limiting traffic flow and determining connectivity.

IETF (Internet Engineering Task Force) - The organization charged with the task of investigating and stardardizing new Internet technology.

IPMP (IP Measurement Protocol) - A protocol developed by Tony McGregor for network measurement. It allows for such measurements as one-way delay, round trip time, and path discovery

IPPM Working Group - A working group within the IETF created to facilitate the development of network measurement protocols.

Mbuf (Memory Buffer) - An mbuf is a common construct used to manipulate packetized data. It allows for efficient manipulation with the least amount of effort.

NIMI (National Internet Measurement Infrastructure) - The National Internet Measurement Infrastructure is a DARPA funded measurement network. It utilizes a daemon (nimid) that collects measurements with tools based on current generation protocols.

NAI (National Analysis Infrastructure) - The National Analysis Infrastructure is a measurement network constructed by NLANR composed of both active and passive measurement probes.

NLANR (National Laboratory for Applied Network Research) - The National Laboratory for Applied Network Research is a NSF funded organization charged with providing technical, engineering, and traffic analysis support for the next generation Internets.

One-Way Delay - The time is takes a packet to travel from its origin to its destination.

RTT (Round Trip Time) - The time it takes a packet to travel from its origin, to a destination, and back.

TCP (Transmission Control Protocol) - A connection oriented, error-correcting protocol used by such applications as telnet and http.

ttl (Time To Live) - A measure of how long a packet is to remain in a forwarding path. If the ttl is exceeded for reasons such a routing loop, the packet is discarded and the sending host is informed via an ICMP message.

UDP (User Datagram Protocol) - A connectionless protocol used by such applications as NTP and the Domain Name System used for translating Internet names to IP addresses.

User land code - Code that is intentionally run without certain permissons and access to parts of the FreeBSD kernel.