Distributed, Agent-Based, Network Fault Diagnosis System

John L. Murdoch

Thesis contributing to hand-in requirements of Napier University in the BEng with Honours Computer Networks and Distributed Systems Degree

School of Computing
Napier University
2003

Authorship Declaration

I, John Murdoch, confirm that this dissertation and the work carried out to complete this dissertation are all of my own achievement.

1) Where work by others has been consulted and used within the context of this dissertation, they are accredited.

2) This dissertation is all my own work.

3) I have acknowledged all main sources of help.

Signed:

Date:

John Murdoch
Matriculation Number: 00034967

Abstract

As networks are getting larger and substantially more complex in companies and institutions, the demands expected of these networks are high. This is indeed the case in environments that rely heavily on e-commerce and the use of technology to do their business. Due to the great reliance upon these networks, the networks must be created, designed, and more importantly, maintained to a high standard. This includes the constant resolution of anomalies found on the network known as faults. To automate the detection and resolution of faults, fault management systems are designed to be implemented upon large networks to find and correct these anomalies.

This report proposes a possible fault diagnosis system, that is distributed across the network utilising agent technologies to gather data from several network nodes and use it to pinpoint potential faults. To implement the backbone of the system, the Simple Network Management Protocol is used to communicate data between both the distributed agents and a central application. At this central application is where the data is processed and displayed to the likes of a network administrator.

After several research of several fault detection methods were analysed, two methods were found to collect data. One of these was to collect data from the hardware of a node, to determine if the hardware was functional, and if the link was intact. Furthermore, to determine routing problems, the Internet Control Message Protocol packets were captured from every host too. A symptom-aggregation technique could then be used to divide the data collated into three distinct model types that would determine if the problem was: client-server based, server/host based or an attack.

The system was then built incorporating all of the above technologies, and tested upon network scenarios to see if the aggregation model technique was successful in isolating the fault type. After this testing, it was then possible to evaluate the success of the system and draw conclusions as to any issues and outstanding work that had not been completed. Finally, the report assessed where the future lay ahead for this work and how it could be improved.

1 Table of Contents

2 Table of Figures

3 Index of Tables

Acknowledgements

I would like to thank Dr William Buchanan first for all his help throughout this project. I would also like to thank the Net-SNMP development team for their helpful tutorials on the use of the Net-SNMP package, for the tutorials on the use of the libPCap packet capturing library and Scyld corporation for technical details and examples about accessing MII registers in GNU/Linux. Finally, I'd like to thank, Brian W. Kernighan & Dennis M. Ritchie, Warren W. Gay, and the late W. Richard Stevens for all their the various books on coding with the C programming language and its application in UNIX-based network environments.

1 Introduction

1.1 Scope, Aims and Objectives of the Project

As time progresses, today's networks are becoming more expansive and complex. As more and more companies and institutions are reliant on these networks for their business activities, the demand for more resilient and efficient network operations is growing. One method of ensuring the network stability is to discover and resolve as many potential network faults as possible. This is where network fault management system plays a key role.

The aim of this project is to create an agent-based system which shall monitor network and host conditions to determine network and host faults, and try and diagnose them. The network and host conditions which are to be monitored, can encompass the entire OSI model; detecting faults situated on the network hardware of hosts and network equipment to the errors generated by host software-based services. Once a sufficient amount of work has been achieved in the construction of the system, it shall be subject to testing and evaluation.

The main stages of the project are:

- **Research –** Investigation of network fault models and on host and server systems, simulation of network fault models, the investigation of mobile and static agent approaches to network fault detection, typical faults.
- **Design –** Involves the design of a model for the collection of data from host agents, and how these integrate with other high-level agents, who are responsible for determining probable faults. It shall also require the design of software for host and network management agents.
- **Implementation –** This will involve the implementation of the key parts of the model, and also elements which shall be used to analyse the performance of the system.
- **Analytical Testing -** This will involve setting up experiments which have known faults, and allowing the system to make judgements on the type of the fault.
- **Conclusions** - This shall present a model of a fault determination system, and outline further experimentation for its implementation in a real-life network.

The main aims of this project were:

- To discover effective ways of detecting faults upon the network and its hosts.
- Deploy a suitable solution for agent-based operation and collation of results.
- Produce a solution that can highlight faults with the data collated from the agents.

2 Theory

2.1 Introduction

A fault is typically defined as an abnormal condition that effects the performance or operation of a system. When applied to the scope of computer network, Kanamaru (2000) presents to us that such abnormal conditions are probably attributed to software and/or hardware error. Such faults can inflict disruption to the users of the local area networks, which can encompass businesses, organisations and customers. It is therefore essential to detect these abnormalities through the process of fault management [Dupuy (1989)]. Fault management is defined by Fuller (1999) as being the area concerned with the detection, isolation and correction of anomalous conditions that occur in the network and maintaining a history of anomalous system behaviour.

There are a number of components of what constitutes the basis of a fault management system:

- **Detection of faults** – the process of collating information on the characteristics and properties of network faults on hosts and determining whether or not these indicators show that the host is deviating from normal behaviour [LaBarre (1991)].
- **Diagnosis of faults** – the process of pinpointing and identification of faults. This can occur immediately at the detection level, but it can also be applied when there is other data available at a time for more accurate diagnosis. This process may also detect the occurrence of many abnormalities caused due to the failure at a single source, so therefore the system may have to filter out such additional indicators so that the fault is not flagged more than once.
- **Correction of faults –** out with the scope of the system at the time of writing, the process of subduing the effect of a detected and correctly diagnosed fault is a welcome idea to those who manage a network. The corrective action may take many forms such as providing alternative resources, rather than attempting to intelligently correct the issue.

2.2 Fault Identification Models

There are several methods and models for identifying faults upon networks, not always directly associated with standard LAN technology, but also with regards to advanced technologies such as ATM. The major method used to indicate faults is through the use of event correlation. The temporal correlation of events is the way in which most fault models work, and involves taking a set of identified symptoms and using these attributes to determine what the problem is.

2.2.1 Rule-based

The use of rule-based systems is perhaps the most widely used and easiest approach to

implementing a fault detection system [Frontini (1991), Hong (1991)]. This generally involves using the knowledge of past faults to generate rules that characteristically identify that anomaly. To detect a fault, these rules are applied to a set of symptoms, where should some symptoms fit a rule or a set of rules, the issue is flagged in some way.

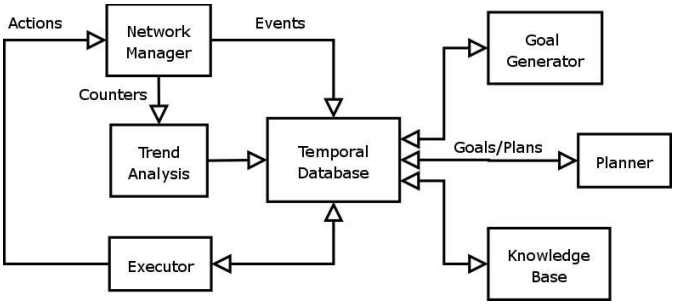


Figure 2.1) - *Joseph's knowledge-based system architecture*

Rule based systems are typically the most common due to the simplicity, but have existed longer than most other approaches. Joseph (1990) proposes such a knowledge-based system, where a fault diagnostic system is built geared towards Manufacturing Automation Protocol (MAP) compliant networks. The system itself contains trend analysis software, a temporal database, a knowledge-base and several components to diagnose and highlight faults. Utilising the trend analysis to reap abnormal measurement data from the network and insert it into the temporal database, the knowledge-base can diagnose possible problems before passing it onwards to other components that can complete the diagnosis and offer a possible solution.

This is a good approach for diagnosing faults, as it appears relatively easy to implement and can provide accurate diagnosis. There are potential flaws however, after some observation. When a potential issue arises, there is a strict definition of what constitutes a fault from the diagnosis system point of view; either a potential fault matches a rule of what a fault should be or it is defined as not being a fault whatsoever. This means that the system cannot intelligently indicate what could be a potential new fault arising. Another flaw derived from this is the human intervention involved to create a new fault profile. To create a new profile, the administrator shall have to detect and pinpoint the fault 'by hand' and insert its characteristics and properties into the knowledge base in the form of a rule. This discrepancy makes these unscalable too and very sensitive to “noise” data. Finally, a minor flaw detected within the system is the potential flagging of duplicate faults: two separate symptoms of a single fault flagged by two individual rules, thus creating two fault indications.

2.2.2 Finite State Machines

The use of finite state machines (FSMs) to model scenarios is a common approach to tackling common computing problems [Rouvellou (1995), Wang (1993)]. As the FSM model is generalised, it can therefore be applied to many different scenarios. Fault detection is such an example.

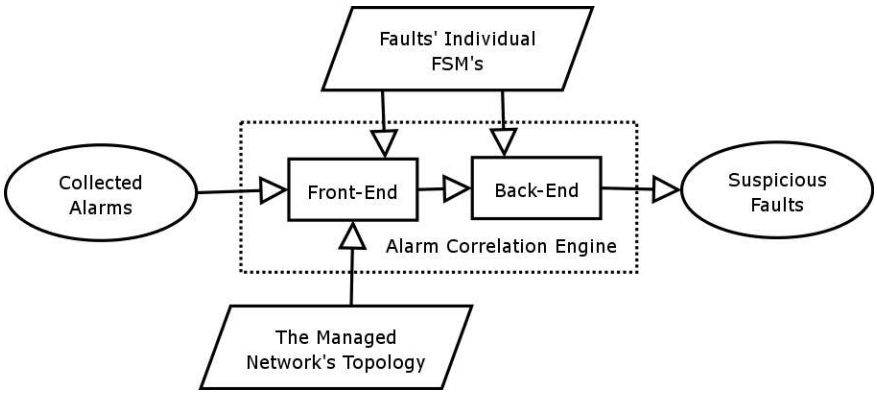


Figure 2.2) - *Alarm correlation engine for fault propagation*

Chao's (2000) LAN fault diagnosis system used FSM-based models to assist with the automated isolation of network faults. The approach focused upon the usage of FSMs to model the concepts of fault propagation behaviour and duration, allowing accurate assumptions as to what possible faults were present upon the LAN during diagnosis intervals. Using these methods, it was possible to correctly correlate anomalies on the network and assess the severity of each. When the prototype system was constructed, within the test environment, the results yielded indicated that the system was performing to high standards.

Another piece of research has implemented FSMs in a different way, applying them to the concept of using multiple observers to detect a fault. Wang (1993), facilitated reduced-state FSMs for each observer, therefore allowing the detection of specified type of faults within a minimum period of time. Each group of FSM observers had approximately two states each, capable of discovering almost every fault present within a system in real-time. In theory, upon violation of the FSM specification, a fault should be detected. There were numerous flaw presented by the author however. Faults such as live-locks and dead-locks could not be detected and faults that involved temporal influences and operational statistics could not be handled through the use of FSMs.

From the outset, the application of FSMs to the concept of fault detection appears promising, as they can be very flexible in the design process. This flexibility tends to yield high performance figures for those who can appropriately assess the correct FSM design to use and can allow the detection of faults that are unforeseen and with incomplete information. FSMs however can become extremely complex, as a intense knowledge of algebra is required to formulate and understand. Lo (2000), also summarises that FSMs are not-suited entirely to real-world situations, due to the high sensitivity to noise generated within some alarms.

2.2.3 Probabilistic Reasoning

Probabilistic reasoning is yet another more common approach to creating a fault detection system [Wang (1993)]. The approach provided by Deng et al (1993), uses a system to collect and filter event information then passes it onto a inference engine. The inference engine produces a fault diagnosis from a algorithm within the inference

engine, in conjunction with the evidence gleaned from the event information and a "priori" knowledge of the current network.

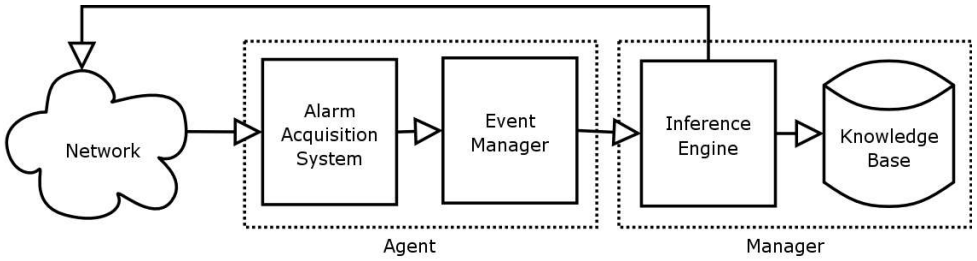


Figure 2.3) - *Deng's network fault diagnosis architecture*

Due to the uncertainties regarding the event information and relationships between these, a probabilistic approach is taken. To implement such an approach, the inference model (i.e. the knowledge-base) used is that of a Bayesian network, also known as a "directed acyclic graph" (DAG). Representing knowledge in such a form is both powerful and concise, as it allows the derivation of an inference algorithm which can work with incomplete data and interactively at the same time. The inference algorithm used in research by Deng works on a "belief" basis, where the function computes its estimate that a component has failed through the evaluation of evidence provided by the event information and inference model. The research has been carried out upon both single and multiple inference models.

This is a relatively good approach to fault diagnosis, as it adapt to certain network models and can cope with incomplete information [Jakobson (1993), Pearl (1997)]. The major flaw however is that if the knowledge is unavailable at the time in which estimates are made (i.e. A newly installed network) and therefore could generate poor estimates of a fault probability. Another minor problem is of building the belief updating algorithm that assesses the probability of the failure, as a lot of research, development and testing needs to be done on the algorithm alone.

2.2.4 Coding-based

Event correlation with the added concept of coding has been subject to some recent studies. The idea behind the coding approach, is that a problem can cause many symptoms upon the network in the form of events, both upon the local device where the fault originated, propagating to other related objects [Kliger (1995)]. A group of symptoms that isolate and identify a fault is given a unique "code" to reference that particular fault. "Decoding" the recorded set of symptoms by calculating which fault has these symptoms as its code.

As stated by Yemini (1996), the analysis of causal relationships among events is essentially the definition of correlation. The partial ordering relationship between events therefore defines those events causality. This can be defined as a causality graph (see Figure 2.4a), where the arrows indicate causality. Using the graph, features like causal equivalence (such as the cycle of events 3,4 and 5), can therefore be depicted as a single event. Also, indirect symptoms (events 7 & 10) can be identified

and pruned from the system. Armed with the knowledge about causal equivalence and indirect symptoms, a corresponding correlation graph can be assembled (Figure 2.4b).

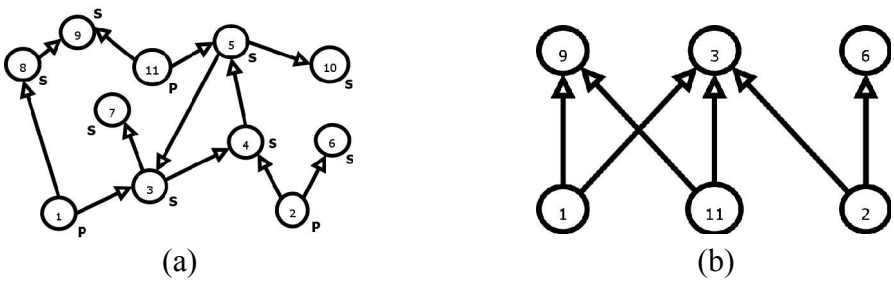


Figure 2.4) - *a) Coding-based causality graph; b) corresponding correlation graph*

The initial research by Yemini (1996) into the field presents a method by which the coding technique can be duly applied. The technique itself is applied in two phases: the "codebook selection" phase and the "decoding" phase. The codebook selection phase involves recreating a correlation graph and giving the information codes, which are simple vectors (e.g. [1,0,1] for problem #1 indicates that symptom #3 and #9 are a result of itself, but #6 is not). Using this process, we can derive that symptom #9 is redundant (as #3 is referenced by not only #1 and #11, but also #2 too), thus reducing the correlation graph. Once the correlation graph has been reduced using these methods, a problem can be identified using a small number of symptoms, rather than a large set. The codebook is therefore a subset of symptoms which can distinctly identify a single problem. A smaller, more optimal codebook therefore means a reduced amount of complex correlation.

	P1	*P2*	*P3*	*P4*	*P5*	*P6*
1	1	0	0	1	0	1
2	1	1	1	1	0	0
4	1	0	1	0	1	0

	P1	*P2*	*P3*	*P4*	*P5*	*P6*
1	1	0	0	1	0	1
3	1	1	0	1	0	0
4	1	0	1	0	1	0
6	1	1	1	0	0	1
9	0	1	0	0	1	1
18	0	1	1	1	0	0

Table 2.1) - *a) Codebook of radius 0.5; b) codebook of radius 1.5.*

The correlation phase facilitates the use of two tables: a correlation matrix which contains all the problems and codes, and the codebook consisting of the symptoms. The codebook can therefore distinctly identify between problems by the 'Hamming' distance between codes. Therefore the radius of the codebook can be calculated as being half the smallest found 'Hamming' distance between codes. A radius of 0.5 indicates that there is great distinction between problems, but a lack of tolerance

towards noise. On the other hand, a radius of 1.5 indicates that up to 2 spurious and lost symptoms shall have no effect on the system.

This set-up appears to be very effective, yet simplistic method to diagnosing faults. Being less complex than designing FSMs or creating probabilistic algorithms, makes it a good choice. It is also very flexible: the codebook can be generated automatically, it can determine if there is enough symptom information for processing and finally the codebook can adapt to the network topology. Effectively, it is simply an optimised rule-based system. However, there may be problems in the implementation of such a system. The automated procedures contained may be simple to look at, but this could perhaps be too much to implement within the time-line of the project.

2.2.5 Symptom-based Aggregation

The aggregation of symptoms may seem to be a rather primitive approach, but it remains a simplistic and viable alternative to approaching network fault diagnostics [Mota (1998), Ohta (1997)]. The technique itself is usually characterised as "divide-and-conquer", as it usually is applied to vast scenarios where the information is abundant in great proportions. Therefore it would seem appropriate that it could be applied on large-scale networks.

Kanamaru (2000) identifies a likely application area where this scheme could be deployed: data packets. When monitored and examined correctly, data packets can provide a whole wealth of information. Using several technologies such as agents and packet monitoring, Kanamaru attempts to create a system to detect faults based on traffic characteristics. The research paper highlights the suitability of monitoring ICMP packets to detect faults and how it usually notifies of problems with network issues such as packet-loss, host reachability and traffic congestion. The sheer abundance of these packets (observed by the author at 1000 per minute on a small network backbone) means that it would be hard to process the packets in one pass.

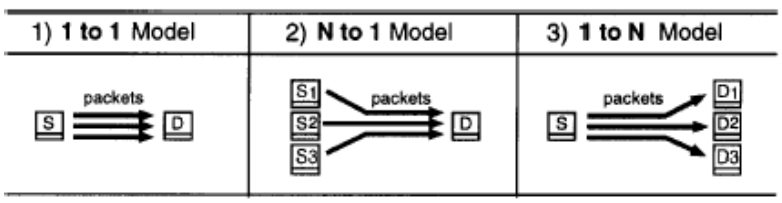


Figure 2.5) - *Packet aggregation models*

The author presents 3 simplistic models to aggregate the ICMP packets: 1:1, 1:N and N:1 in relation to the source host and destination. Each of these models acts in unison as a filter, therefore dividing up the packets into groups of "remarkable symptoms" or passing the remainder recursively back through the "filter" until there is no enough packets left to constitute a remarkable symptom. Analysing the abundance of each ICMP packet type (therefore symptom) shall usually characterise a specific fault.

This system is exceptionally easy to understand due to the lack of complexity within each of the models and is perhaps the most simplest, yet effective fault detection

method so far. The use of recursion also allows the faults to be scaled in significance, therefore being assistance in determining which problem is most disruptive. The simplicity involved and the level of feedback obtained makes this technique attractive to implement, even when in comparison to the "Coding-based" approach described above.

2.2.6 Diagnostic Reasoning (Expert Systems)

Diagnostic solutions is more commonly associated with the development of what is known as 'Expert Diagnostics' [Reggia (1983)]. According to Fuller (1999), the techniques commonly associated with expert systems and both network management include:

1. Representation and application of heuristics for the solution of mathematically intractable problems.
2. Reasoning in the absence of complete information or the presence of ambiguous results or incorrect data.
3. Automation of complex, situation-dependant procedures subject to multiple constraints that are typical of standard operating procedures developed as contingency response for emergency scenarios.

Diagnostic reasoning has some overlap with rule-based systems, as they can be implemented using rules, but instead have a 'reasoning' approach to make the rules more flexible to accepting border-line faults and the occurrence of new faults. However, due to the complexity involved (such as dedicated artificial intelligence as deployed by Inder(1988)) in the creation of individually specialised functions, diagnostic expert systems can be discarded as an implementable approach in this scenario.

2.3 Agent Technologies

Software agents are programs which can take action, collate data and can flag events from remote machines. These agents can therefore be used for successful implementation of fault detection and performance management, essential for the creation of a fault management system. Agents can be built in one of three ways: coded entirely from scratch by the developer, using a specialist integrated development environment (IDE) for creating agents or by using pre-built, customisable agents. To interact with the agents are usually complimented with a manager; a program which can deploy, manage and observe agents.

As stated by Muller (1997), the agent-manager concept goes hand-in-hand with most network management protocols, especially the standardised Simple Network Management Protocol (SNMP) [Case (1990), Stallings (1999), Mansfield (1992)]. SNMP is a highly popular protocol to manage most modern networks, as it is available or implemented in a variety of network hardware such as bridges, hubs, routers and switches. SNMP defines that interaction can occur between the agent and the management software in two ways: by the manager polling the agents for information or by the use of traps or triggers on the agents which can alert the

management station.

SNMP can assist in the analysis of performance and fault management, the following is a brief summary of possible application areas:

- **Network Trending** – The ability to analyse or visualise the difference between 'normal' and 'abnormal' behaviour and to ensure that any properties of the network are within Quality of Service (QoS) thresholds.
- **Network Usage** – Identifying what traffic types or applications are utilising the network and at what times.
- **Client/Server Performance** – Discovering servers that may be overloaded or which clients may be 'hogging' the resources of a server.
- **Data Correlation** – Being able to use multiple sources of information together to determine the source of performance and faults. An example is identifying a network performance issue by observing what applications were running at the time and how many errors were observed.
- **Packet Interrogation** – Isolating and observing a particular stream or session of traffic which may be causing network issues.
- **Error Generating Nodes** – Track down nodes which are generating large volumes of errors on the network, possibly impacting the performance of the network.
- **Fault Notification** – Gives the agents the possibility to flag faults automatically or pro-actively before they happen.
- **Automated Resolution** – The possibility of the agent to handle simplistic or trivial issues which may be degrading network performance.

From the outset, SNMP appears to be the sole, most common, standardised protocol to create agents and management station software. On the other hand, designing and creating a network management protocol seems to be a daunting task, as it would be hard to implement support for the likes of hardware devices and obscure operating systems.

2.4 ICMP Data Standards

As part of any IPv4 or IPv6 implementation, the Internet Control Message Protocol (ICMP) forms an important and integral part of the IP protocol as a whole. As stated by Stevens (1998), ICMP is normally used to communicate error messages between IP nodes, both routers and hosts, but it is occasionally used by applications.

ICMP has formed part of the research as it plays an important role if packet aggregation is to be used in the fault diagnosis system. ICMP provides a method of returning messages back to a host or an application when an error occurs, so therefore the system shall be constructed to collect any ICMP packets a host may receive and use these to determine a faulty node or link. ICMP has a distinct format for each data packet, as stated by the RFC from Postel (1981) declaring ICMPv4:

0 *7*	*8* *15*	*16* *31*
type	code	checksum
Remainder dependant upon ICMP type and code		

Table 2.2) - *ICMPv4 message format*

The 'type' field consisting of 8 bits, represents the type of the ICMPv4 error message that is being reported. The additional field, 'code', provides additional information, giving greater granularity as to the potential error profile. The checksum provides an added a way to ensure that the data provided in the ICMP header and content is intact and correct.

Type	***Code***	***Description***	***Type***	***Code***	
0	0	echo reply	4	0	Source quench
3	0	Network unreachable	5	0	Redirect for network
3	1	Host unreachable	5	1	Redirect for host
3	2	Protocol unreachable	5	2	Redirect for type-of-service and network
3	3	Port unreachable	5	3	Redirect for type-of-service and host
3	4	Fragmentation needed but DF set	8	0	Echo request
3	5	Source route failed	9	0	Router advertisement
3	6	Destination network unknown	10	0	Router solicitation
3	7	Destination host unknown	11	0	TTL equals 0 during transit
3	8	Source host isolated	11	1	TTL equals 0 during reassembly
3	9	Destination net. administratively prohibited	12	0	IP header bad
3	10	Destination host administratively prohibited	12	1	Required option missing
3	11	Network unreachable for TOS	13	0	Timestamp request
3	12	Host unreachable for TOS	14	0	Timestamp reply
3	13	Communication administratively prohibited	15	0	Information request
3	14	Host precedence violation	16	0	Information reply
3	15	Precedence cut-off in effect	17	0	Address mask request
			18	0	Address mask reply

Table 2.3) - *ICMPv4 messages*

Above is an example of all the errors that can be generated via ICMP. Some of the larger type categories can be summarised as follows:

- Type 3 – Destination unreachable
- Type 5 – Redirect
- Type 11 – Time exceeded
- Type 12 – Parameter problem

This system therefore allows an observer to find and diagnose possible routing errors

within the system. Strangely enough, using ICMP packets is a classic method used for denial of service attacks (ping flooding) and for port scanning, therefore any system using ICMP packet aggregation can be possibly detect malicious activity occurring within and/or directed from outside the network.

2.5 Conclusion

Concluding the research, it appears there are many avenues of opportunity for fault detection, comprising of several methodologies for fault detection. However most of these methodologies are complex and distinctly hard to implement due to the unique mathematical algorithms or artificial intelligence used. An exception to this characterisation on the other hand, is the use of symptom-based aggregation or coding-based techniques. The coding-based method however, needs to create a knowledge-base of information, and although this is relatively easy to understand and simple to do on paper could be more complex and cumbersome than the recursive, simplistic methods used in symptom-based aggregation.

The symptom-based aggregation technique outlined by Kanamaru, also provides a unique method of utilising ICMP, a standard implementation of delivering error messages. By intercepting and finding the most common types of ICMP packets being distributed around the network [Waldbusser (1995), Perkins (1999)], this can determine the most disruptive error location and cause. Through the use of distributed agents too, the mass collection of ICMP packets is possible. Agents could thereafter capture these ICMP packets, where they can be collated by a central management station which could preform aggregation techniques to find the most disruptive forms.

There was however a distinct absence of research data available on most subjects, most notably fault modelling and details of faults. A fault list was compiled by researching from the Internet, but it proved too ambiguous when attempting to classify the data appropriately. Therefore instead of using faults, it was decided to focus on areas where faults could be determined using ICMP and MII.

3 Methodology

3.1 Introduction

This chapter discusses the design of a distributed, network fault diagnosis system with the use of agents and central management station software. Utilising this configuration, it should be possible to collate data from remote nodes and use this to determine where a fault is. Here are the steps involved in the operation of the system to building a normal, generic SNMP system [Mansfield (1992), Ohta (1996)]:

1. Gather data from node where agent resides.
2. Agent retrieves and/or stores gathered data.
3. Agent sends data to Management station upon request.
4. Management station brings data from agents together.
5. Management station interprets data to give it meaning.
6. Management station displays data for human analysis.

3.2 Distributed Agents

The system itself shall be designed adhere to SNMP standards, utilising the agent-manager concept identified by Muller (1997). Therefore the agents shall exist upon a target host, where they can be polled for information. The information can be available in two forms: a value or a table of values. The individual values shall represent two things: the status of something on a host and a setting, perhaps a threshold or a restriction. The table however, shall be a two-dimensional array, where various elements of information can be held about a specific event along the horizontal axis and the vertical axis indexes each event.

The management station is used to retrieve information from each of these agents tables and/or values, where the agent set to be polled is from a predefined host list. The system may also have the capability to set thresholds to lower the memory usage of the agents or to increase the processing/storage capacity should the agent be overwhelmed with too much information that the threshold makes information held redundant quickly.

3.3 Data Acquisition

There are two types of data that be considered within the system for fault diagnosis: ICMP packet data and network card status. Both of these can be used to differentiate between hardware, software or network media faults.

3.3.1 Network Media Diagnosis

Interfacing with the network card (NIC) directly can yield a useful source of information. The information about the current status of NIC is held within registers. On a standard 10/100-compliant Ethernet NIC, this is held on what is known as a

Media Information Interface (MII) registers [Becker]. These registers contain information about the link between itself and the other end-point on the attached Category 5 (or 6) cable. Such information includes connection speed, collision detection and most importantly whether a link is present. This link is detected by what is known as a network 'heartbeat'.

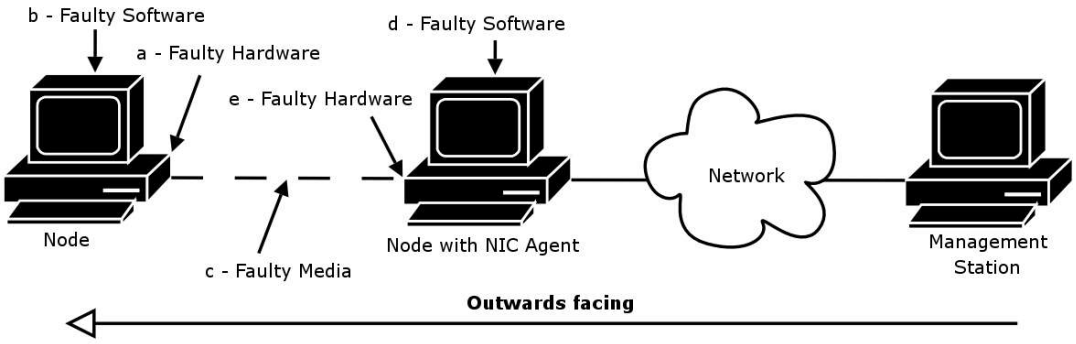


Figure 3.1) - *Diagram of NIC agent fault detection*

Interfacing with the NIC MII, it should be possible to read the registers of a connected node which has an 'outward' facing NIC and see if the connection is alive. Here is the three obtainable fault scenarios from the NIC agent node:

1 The local node has a software problem, if we cannot set-up a TCP/IP socket.
2 The local node has a hardware problem, if we cannot do I/O operations through a TCP/IP socket to the MII registers.
3 The host has an indeterminable connection problem, being one of the following:
  3.1 The node on the other end has a hardware fault.
  3.2 The node on the other end is experiencing a software fault.
  3.3 The interconnecting media is faulty

Therefore it is possible to determine a route to a host which is unreachable, possibly even the agent pinpointing the error on itself. If there was two routes between the manager and the 'faulty' host, if both connections were faulty, it would be more likely to be a software issue, ruling out a cable error and decreasing the probability of a hardware fault (although this is still likely). On the other hand, if one connection was working, but the other faulty, it would be a safe assumption that a cable fault would be the most likely fault, followed by a hardware fault, and it would be possible to dismiss a software fault if the node has a system-wide implementation of TCP/IP.

Therefore, the Network Media agent shall be designed as a dedicated agent which can respond to requests and take a 'reading' of the MII registers on the NIC. The data regarding the state or the accessibility of the MII shall be relayed back to the Management Station perhaps in the form of an integer code for efficient network usage.

3.3.2 ICMP Data Logging

As stated in the literature review, ICMP packets were processed by Kanamaru (2000)

using a packet/symptom aggregation technique. For the network fault diagnosis system, this technique shall be emulated, with the exception of how it is implemented. The agents themselves should be a dedicated, autonomous piece of software with the sole goal of capturing ICMP packets. Once constructed, this shall be a small, efficient, more distributed method of monitoring multiple hosts without the use of third-party software.

The ICMP data should be essentially 'captured' from the network card, possibly using the PCap library which is available for most programming languages and platforms. This allows the system to go into a promiscuous mode where all the packets going across the interface can be observed. Upon the capture of a packet, the packet must be stored in a way that is readily servable to the Management Station software and is formatted into an appropriate table.

Time	*Source IP*	*Dest. IP*	*ICMP Type*	*ICMP Code*
22/10/2003 23:23:23	192.168.0.1	192.168.0.2	2	0
22/10/2003 23:25:24	192.168.0.3	192.168.0.2	8	1
23/11/2003 00:01:56	192.168.0.3	192.168.0.2	5	0
...	...	...	...	...
dd/mm/yy hh:mm:ss	IP Address	IP Address	Integer	Integer

Table 3.1) - *ICMP SNMP storage example*

Table 3.1 depicts an example table of how the data could be stored. The five fields depict information that is required for making it possible to correlate data together in the future and to keep information down to a low level:

- **Time** – Useful for noticing trends and correlating events with other hosts. It can also be used to eliminate old data so as to keep the agent to a small level, possibly removing old data prematurely so that room can be made for new packets, should the table overflow.
- **Source IP** – Where the data is coming from. This host could be generating a lot of errors, (server with software problems, overloaded router and Denial of Service (DoS) attacker), so it is essential that we know the source address so we can see how much errors each specific host is generating.
- **Destination IP** – The IP of the current host. Perhaps irrelevant at the time of writing, but useful if there are more than two network interfaces installed on a machine.
- **ICMP Type** – What category does the ICMP packet come under? It may indicate a unreachable node or nodes or a spate of 'ping' packets from an attacker.
- **ICMP Code** – Presuming that we know what type of ICMP errors are occurring and what scope they are in, this field may show a further level of error diagnosis. I.E Type 8, code 0 packets indicate that the node could be on the receiving end of a DoS attack. Type 8, code 1 may indicate that a host is 'spoofing' the nodes address, hence we are receiving the replies to the DoS.

Once this data has been collated, it should be possible to sort the data, cleanse away old data and possibly overwrite the oldest of data in an emergency situation. This should be a prime consideration, as if the agent is to pass table-arranged data across the SNMP protocol, the information must be of significance and of smallest proportions.

3.4 Packet Aggregation Models

As identified by Kanamaru (2000), there are three basic models which constitute complete packet aggregation:

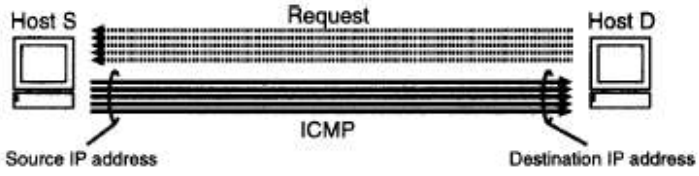


Figure 3.2) - *1:1 example fault*

- **One to One –** This model categorises a specific client-to-server failure. This means that the fault identified does not affect any other hosts or servers. This fault however may cause other symptomatic faults in its wake, such as the server slowing down as a client "hogs" its resources. The proposal is to check that the destination and source IP's are the same for each ICMP packet type. The more packets that have the same matching destination, source, type and code should increment a value, and be inserted into a table. The highest of the values within this table shall be the most dominant of the aggregation type.

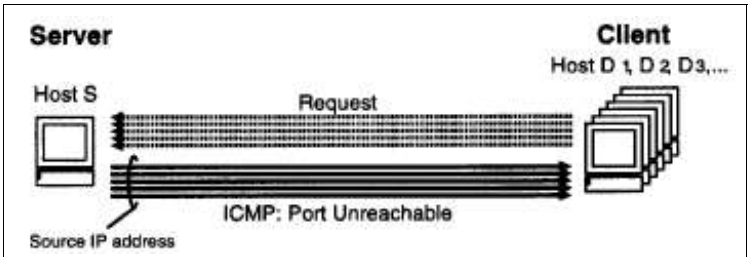


Figure 3.3) - *1:N example fault*

- **One to Many –** This model characterises a fault where node causes faults to be observed by numerous hosts. Typically this should be observed when multiple clients connected to a central server, receive notification packets when a the server fails. The proposal is to check for ICMP packets that have the same source IP. Therefore if the packet has the same source IP, type and code, then the packets that have the same details should be inserted into a table once along with the number of occurrences found. The highest of the values within this table shall be the most dominant of the aggregation type.

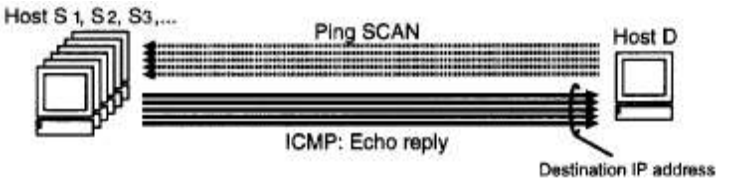


Figure 3.4) - *N:1 example fault*

- **Many to One** – Finally, this model can detect malicious activity, as several hosts start emitting error packets to a central point. This can be seen as a distributed denial of service, a single host attempting a spoofed denial of service and a host scan. The proposal is to check for ICMP packets that have the same destination IP. Therefore if the packet has the same destination IP, type and code, then the packets that have the same details should be inserted into a table once along with the number of occurrences found. The highest of the values within this table shall be the most dominant of the aggregation type.

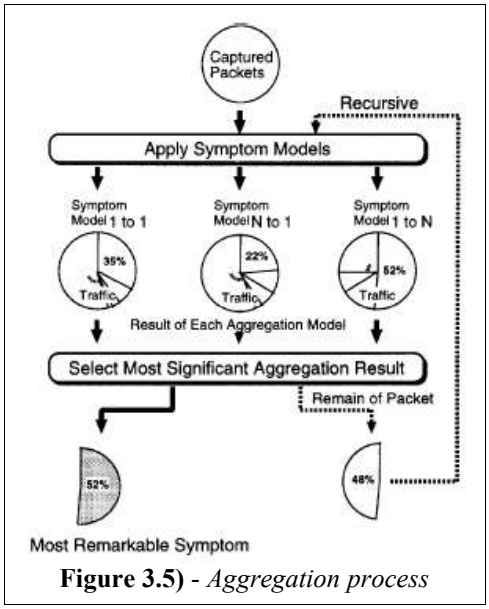


Figure 3.5) - *Aggregation process*

Once all the types have been counted, there should be one candidate from each model. These candidates should be compared and biggest of the three models wins and becomes the main "fault" or "most remarkable symptom" seen on the network.

3.5 Conclusion

It has been determined that it should be an obtainable task to create a simple network fault diagnosis system using several technologies together. SNMP will provide the groundwork for the system, being a method of communicating between both client agents and a central server or what is known as a "management station" application. This can be used to pass both tables and small values to and from the agents.

Secondly, there are two methods to collating data on faults: network media detection and ICMP packet collection. The network media detection method can detect a fault on three levels: hardware, software and media. Therefore from one side of a node, it should be possible to detect if itself is the cause of a fault. If this is not the case, then the focus can shift onto the media or the other node, depending on the media connection status.

The other remaining method is ICMP packet capture. Although the packet capture alone is fairly useless, when large amounts of data are collated together, it should be possible to aggregate this data into groups of three models. The largest of these three

groups should therefore be the most disruptive, and therefore the ICMP can be interpreted by the user to explain what the exact problem is.

4 Implementation

4.1 Introduction

This section details of how the system in Chapter 3 was implemented through the use of SNMP. It also outlines how data was collected, stored and retrieved via SNMP and through the organisation of a user-defined MIB table. Short samples of code, screen-shots and diagrams of the system shall be used to convey how the system meets the criteria defined in chapter 3.

4.2 System Requirements

To choose an operating system to build a network fault diagnosis system upon, there was no immediate candidates which sprang to mind. Given enough time and research, it was determined that the GNU/Linux operating system seemed the most appropriate due to its openness and portability. As GNU/Linux complies to the GNU General Public License, this gave the project the ability to work upon a operating system that makes the original source code for the entire freely available, therefore making integration easier as the specifications for the inner workings for operating system could be easily obtainable. Another benefit would be the fact that GNU/Linux provides a healthy amount of software which can be modified. This would enable the project to be built using existing software source code, therefore avoiding the process of "re-inventing the wheel".

As the system choice was Linux there was several choices for a programming language: C, C++, Java, Perl, etc. The C programming language was settled upon as the best suited software to write a fault diagnosis system, as it was much more appropriate for the likes of system programming compared to the application-development oriented Java. This meant it would be much easier to access the lower levels of the OSI model, such as the network layer. C was also the lowest common denominator in terms of the languages available, as the Linux kernel was predominantly written in C.

Although it would have been possible to write an SNMP-compliant protocol, agents and management station, the time constraints of the project meant that there was very little time available to re-invent what was already freely available. There were a few SNMP solutions available for GNU/Linux, but the particular software which stood out was Net-SNMP (formally known as UCD SNMP). Net-SNMP has the advantages of being cross-platform, available for the C programming language and had support for the creation of both agents and applications, with a wide variety of features too. Besides development, the package also comes with the source for a default SNMP daemon that can run on hosts, supporting a large number of sub-agents.

The following is a summarised specification with noted alternatives and future possibilities:

- GNU/Linux operating system, Kernel 2.4 or 2.6. Possible porting opportunities in the future to port most of the C code to Microsoft Windows.
- Intel x86-compatible computer system. This may compile under other various architectures with the ability to run GNU/Linux, such as: SunOS, Sun Solaris, IBM AIX, Apple Macintosh, etc.
- 10/100 Ethernet card, where the card is fully supported within GNU/Linux, giving full MII register access [Becker].
- Net-SNMP, a set of libraries for the creation of C programs with SNMP support, along with a central daemon which acts as a master, hosting agent for smaller, user-defined sub-agents.
- LibPCap, a network packet capture library for C programs [Jacobson].

4.3 SNMP Daemon Installation Upon Hosts

As explained previously, the Net-SNMP package comes with a pre-coded, SNMP daemon called 'snmpd'. This program is usually run upon boot as a service not unlike web server or distributed file system software. Therefore, the SNMP daemon is able to collate and serve information from a host while it is on-line, hosting sub-agents to retrieve information. For the network fault diagnosis system (NFDS) to work correctly, this software must be deployed upon all target machines that are to be monitored.

Before the daemon can be deployed and if custom sub-agents are to be used, the program must be compiled with shareable module support, as some sub-agents can take the form of a pluggable module. This is the case for the network media diagnosis module explained later.

```
syslocation  "Home or Napier"
syscontact  "John Murdoch - 00034967@napier.ac.uk"

##############################################################################
# SECTION: Access Control Setup
# SNMPv3
rwuser  private
rouser  public noauth

# SNMPv1/SNMPv2c
rocommunity  public
rwcommunity  private

com2sec public        default public
group  public v1      public
group  public         v2c    public
group  public usm     public
view   all    included       .1
access public ""      any    noauth exact all none none


##############################################################################
# SECTION: Trap Destinations

trapsink  spadge.localnet
trap2sink  spadge.localnet
informsink  spadge.localnet
trapcommunity  public
authtrapenable  1
```

Once the program has been installed (see Appendix A for compilation, installation and configuration details), a customisable configuration file must be created which governs how the daemon behaves. Obligatory details such as location, contact information, and user access control to the daemon must be entered for the daemon to be securely implemented. Other details that may be configured, is how traps (agents that can alert the management station) and how sub-agents can interact with the daemon. Once the daemon has been correctly compiled, installed and configured, can the agents be added. The following is the configuration file used in this implementation:

4.4 Creation of the Agents

The agents were constructed using Net-SNMP, a free open-source toolkit and libraries for making SNMP-enabled applications and agents. The toolkit provides a sample daemon along with sample applications and plug-ins. Rather than construct a daemon from scratch, the use of plug-ins appeared to be the simplest method of creating agents without the overhead of coding a new daemon, as the bundled one could be used instead. There are many types of plug-in styles too, but these will be discussed according to each agent. As the user can create their own agents, the Net-SNMP toolkit allows the developer the power to create their own MIB (Management Information Base), which shall be discussed next.

4.4.1 SNMP MIB Table

SNMP uses what is known as a management information base table to hold a index of the entire SNMP functions of the daemon and any other additional modules. The layout is defined in a tree structure, divided into subject area. The Net-SNMP libraries when compiled into a program can translate numeric object identifiers (OIDs) into textual object identifiers using MIB description files. The Net-SNMP package contains some common MIB definitions used in common network hardware and the Net-SNMP bundled daemon and plugins.

To implement custom plugins however, a user-defined MIB must be constructed around the plugins, so the values and settings can be accessed. As seen in figure 4.1, a custom MIB consists of several elements:

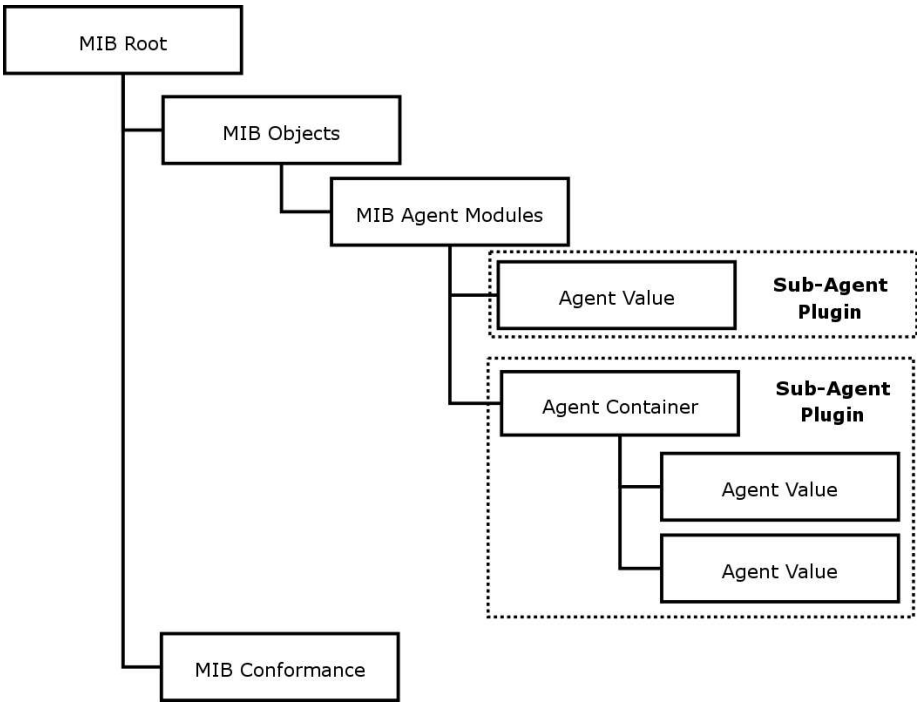


Figure 4.1) *- Typical user-defined MIB layout*

- **Root** – this defines where the user-defined MIB fits into the global MIB and the contact details, etc.
- **Object Container** – Container for any accessible object whether they are located on a daemon or a plugin of sorts.
- **Agent Modules Container** – Container for agent modules.
- **Agent Value** – A value which can be retrieved from the plugin or daemon.
- **Agent Container** – Container for keeping many related agent values together.
- **Conformance** – Required for maintaining consistency.

The following is the resultant incomplete MIB for housing the two Agents for the system. Note that the agent definitions are incomplete and shall be explained later:

```
1. NETWORK-FAULT-DIAG-SYS DEFINITIONS ::= BEGIN
2.
3. -- IMPORTS: Definitions from other mibs
4. IMPORTS
5.      netSnmpExamples                                 FROM NET-SNMP-EXAMPLES-MIB
6.      MODULE-IDENTITY, OBJECT-TYPE,
7.      Integer32, Unsigned32,
8.      TimeTicks, Counter32, Counter64,
9.      IpAddress                                       FROM SNMPv2-SMI
10.     TimeStamp, DateAndTime                          FROM SNMPv2-TC
11.     MODULE-COMPLIANCE, OBJECT-GROUP                 FROM SNMPv2-CONF;
12.
13.-- A brief description and update information about this mib.
14.netFaultDiagSysMIB MODULE-IDENTITY
15.     LAST-UPDATED "200310050000Z"
```

```
16.     ORGANIZATION "Napier Univeristy"
17.     CONTACT-INFO "email:  john.murdoch@weejoker.demon.co.uk"
18.     DESCRIPTION  "Distributed, Agent-based, Network Fault Diagnosis System"
19.     ::= { netSnmpExamples 4 }
20.
21.-- Define typical mib nodes, like where the objects are going to lie.
22.nfdsMIBObjects     OBJECT IDENTIFIER ::= { netFaultDiagSysMIB 1 }
23.nfdsMIBConformance  OBJECT IDENTIFIER ::= { netFaultDiagSysMIB 2 }
24.
25.-- define 1 object container
26.nfdsAgentModules   OBJECT IDENTIFIER ::= { nfdsMIBObjects 1 }
27.
28.-- This is the single object value
29.nfdsSenseLink OBJECT-TYPE
30.     DESCRIPTION
31.     "This is an object that returns the current link status of the
32.     network card."
33.     ::= { nfdsAgentModules 1 }
34.
35.nfdsIcmp OBJECT IDENTIFIER ::= { nfdsAgentModules 2 }
36.
37.END
```

Visible in the above listing is the containers and how they are linked. Module Identity indicates the root of the MIB (named 'netFaultDiagSysMIB')and its parent which it links to in the global MIB tree is 'netSnmpExamples', where it assumes the fourth entry in the tree. 'nfdsMIBObjects' and 'nfdsMIBConformance' are the first and second children from the root. 'nfdsAgentModules' is a container within 'nfdsMIBObjects', where there is a plugin container for the ICMP table called nfdsICMP and an object-type called nfdsSenseLink. This shall be explained further in the next chapter.

4.4.2 Network Media Diagnosis

The aim of this module is to allocate a value to the current status primarily of a network link, but also the hardware and software which constitute an endpoint of a link. A prototype was created for the first instance of development of the agent for two reasons: to assess the potential for read the MII registers on a NIC [Becker] and to test the functionality without the complications of using SNMP in between.

```
1. if ((ioskt = socket(AF_INET,SOCK_DGRAM,0)) < 0)
2. {
3.      perror("socket");
4.      exit(-1);
5. }
```

The above piece of C code determines whether or not sockets work, therefore indicating if the protocol stack is functional. This provides the detection for software failure, as if the generation of a socket produces an error value (less than zero), the 'if' statement is true, an error is generated and the program exits.

```
1. if (optind == argc) { iface = "eth0"; }
2. else
3. {
4.      spp = argv + optind;
5.        iface = *spp++;
6. }
7. if (iface == NULL) { iface = "eth0"; }
8. printf("Using network interface: %s\n", iface);
9. strncpy(ifr.ifr_name, iface, IFNAMSIZ);
```

This piece of code processes the input from the prototype command line which should consist of the name of a network interface. If the name is not found or there is no arguments passed, this defaults to the first, most common Ethernet interface name on GNU/Linux, 'eth0'. This interface name is then transferred into an interface structure called **`ifr`** translated into the **`ifr_name`** variable.

```
1. int check_iface_ioctl(int ioskt, char *iface)
2. {
3.      unshort *data = (unshort *)(&ifr.ifr_data);
4.              data[0] = 0;
5.
6.      if (ioctl(ioskt, 0x8947, &ifr) >= 0) { ioctl_type = 1; }
7.      else if (ioctl(ioskt, SIOCDEVPRIVATE, &ifr) >= 0)
8.              { ioctl_type = 0; }
9.              else
10.             {
11.                     printf("MII access on %s failed: %s\n", iface, strerror
   (errno));
12.             (void) close(ioskt);
13.                     exit(-1);
14.     }
15.}
```

The previous interface name is then passed to the **`check_iface_ioctl`** function along with the previously created socket. This function then checks if it is possible to do input/output (I/O) operations upon the network card. The first 'if' statement checks if it is possible to do the newest form of I/O on the MII registers on the NIC, the second 'if' does the older form. If no I/O can be performed, then the program exits.

```
1. int get_connection_status(int ioskt)
2. {
3.      unshort *data = (unshort *)(&ifr.ifr_data);
4.      // First piece of data is the physical ID
5.      unsigned phy_id = data[0];
6.
7.      // 0x0004 indicates that there is no cable attached
8.      if ((read_mii(ioskt, phy_id) & 0x0004) == 0)
9.              printf ("Cable disconnected from NIC\n");
10.     else
11.             printf ("Cable connected to NIC.\n");
12.         return 0;
13.}
14.
15.int read_mii(int ioskt, int phy_id)
16.{
17.             unshort *data = (unshort *)(&ifr.ifr_data);
18.             // First piece of data is physical ID, next is the
19.             data[0] = phy_id;
20.             data[1] = 1;
21.
22.             if (ioctl(ioskt, ioctl_type ? 0x8948 : SIOCDEVPRIVATE+1, &ifr) < 0) {
23.             printf("MII register read on %s failed: %s\n", ifr.ifr_name, strerror
   (errno));
24.             return -1;
25.             }
26.             return data[3];
27.}
```

Finally the established socket is used to read the MII contents by the

`get_connection_status` function. The variable **`phy_id`** shall hold the physical address, which is located in the first two bytes of data from the interface structure (**`ifr`**). The **`read_mii`** function takes the socket and the physical address, puts these values in a new interface structure (via **`data`** pointer), then attempts to read a byte at position 0x8948 via I/O. The connection information is then accessible from the fourth position within the **`ifr`** structure (via the **`data`** pointer), so therefore it is returned to the **`get_connection_status`** function from where it was called initially. Using a bit-mask on bit number four of the data returned contains the link connection information: if this bit is set at 0 (false) then the cable is disconnected, otherwise if the bit is 1 (true), then the cable is connected.

The SNMP plugin wrapper chosen to encapsulate the functions above was the most lightweight, portable and easiest to create. Creating the sub-agent as a dynamically loadable shared module, meant that it was possible to build a plugin for the Net-SNMP daemon, but without having to recompile the main daemon to install it. This meant that a architecture-specific binary module could be created once and then deployed upon several hosts, providing that the systems were homogeneous in terms of processing and memory addressing, etc.

```
1. void init_nfdsSenseLink(void)
2. {
3.      static oid nfdsSenseLink_oid[] = { 1, 3, 6, 1, 4, 1, 8072, 2, 4, 1, 1, 1, 0 };
4.
5.      netsnmp_register_read_only_int_instance("nfdsSenseLink",
6.                                              nfdsSenseLink_oid,
7.                                              OID_LENGTH(nfdsSenseLink_oid),
8.                                              &nfdsSenseLink,
9.                                          link_status_handler);
10.}
```

The above function is what the SNMP daemon calls to initialise the module. The OID (object identifier), correlates what is seen in the MIB tree. The numbers ranging from left to right showing the nodes in the tree where it resides. The second line of the function registers and defines a read-only value associated with the agent, consisting of:

- Name of the object within the tree.
- Object Identifier
- Size of the object identifier.
- Global variable which contains the object value/data to be returned.
- Handler function to be called when a request for the object is received.

```
1. int
2. link_status_handler(netsnmp_mib_handler *handler,
3.                     netsnmp_handler_registration *reginfo,
4.                     netsnmp_agent_request_info *reqinfo,
5.                     netsnmp_request_info *requests)
6. {
7.     switch (reqinfo->mode) {
8.     default:
9.             requests->delegated = 1;
10.            snmp_alarm_register(1,
11.                                0,
12.                                return_link_status,
13.                                (void *)
14.                                netsnmp_create_delegated_cache(handler,
15.                                                               reginfo,
16.                                                               reqinfo,
17.                                                               requests,
18.                                                               NULL));
19.            break;
20.    }
21.    return SNMP_ERR_NOERROR;
22.}
```

The handler function for the module (as shown above) processes the requests. As there are several types of SNMP request modes, these must be handled via a switch statement. We must however use request delegation, answering the request at a later time. This has to be done because one must poll the network interface for its current status, then return the new value back to the caller, rather than send back a the current value before it is updated. The checking of the software, hardware and link status should take no more than a second, therefore a SNMP**_alarm** is created which times out one second later. The alarm function consists of 4 fields:

- the time delay in seconds
- is the request repeated at a regular interval? (boolean)
- the function to call
- the creation of a cache of request informant so that a request callback can be performed

```
1. void return_link_status(unsigned int clientreg, void *clientarg)
2. {
3.     netsnmp_delegated_cache *cache = (netsnmp_delegated_cache *) clientarg;
4.     netsnmp_request_info *requests;
5.     netsnmp_agent_request_info *reqinfo;
6.     cache = netsnmp_handler_check_cache(cache);
7.     if (!cache) {
8.             snmp_log(LOG_ERR, "illegal call to return current link status\n");
9.             return;
10.    }
11.    reqinfo = cache->reqinfo;
12.    requests = cache->requests;
13.    requests->delegated = 0;
14.    switch (cache->reqinfo->mode) {
15.            case MODE_GET:
16.            case MODE_GETNEXT:
17.                    get_link_status("eth0");
18.                    snmp_set_var_typed_value(cache->requests->requestvb,
19.                                             ASN_INTEGER,
20.                                             (u_char *) & nfdsSenseLink,
21.                                             sizeof(nfdsSenseLink));
22.                    break;
23.    }
24.    netsnmp_free_delegated_cache(cache);
25.}
```

`Return_link_status` is then called by the SNMP alarm, extracting the contents of the cache. The cache is then checked for inconsistency, to ensure that the request has not timed-out for example, otherwise an error is returned if the cache does not pass the test. The original request pointers are then resurrected within the function, and the request is now no longer marked as delegated, as it is about to be processed. Presuming the request is a SNMP "GET"or "GETNEXT" commands (not unlike the behaviour of the HTTP protocols "GET"), then it we should get the link status. The **`get_link_status`** function is exactly the same as the prototype version, with the exception of the use of the integer **`nfdsSenseLink`** throughout instead of exit. At each point of failure (sockets, I/O, or MII readings), a value is assigned to this variable where the function would normally exit (this can be seen in the appendices). When the **`get_link_status`** function has completed its task, there should be a value within the **`nfdsSenseLink`** integer. This is then committed using the **`SNMP_set_var_typed_value`** call and the request should be sent to the caller. Before the function exits however the cache entry must be cleared, as can be seen in the last line.

To incorporate the integer into the MIB, the following is the complete **`nfdsSenseLink`** entry:

```
1. nfdsSenseLink OBJECT-TYPE
2.     SYNTAX      INTEGER {
3.      fault(0),
4.      disconnected(1),
5.      connected(2)
6.     }
7.     MAX-ACCESS  read-only
8.     STATUS      current
9.     DESCRIPTION
10.     "This is an object that returns the current link status of the
11.      network card."
12.    DEFVAL { 1 }
13.    ::= { nfdsAgentModules 1 }
```

Note that all the possible integer values are enumerated, the value itself is set as read-only and the default value is set as 1.

To compile the plugin easily, a simple 'Makefile' was created. This handles the compilation issues such as compiler flags, libraries to compile in and compilation dependencies. Simply issuing "make" from the command line whilst in the source directory, compiles the module appropriately (see Appendix for Makefile).

Finally, the binary SNMP shared-object plugin (**`nfdsAgentPluginObject.so`**) was copied to the '**`/usr/lib/SNMP/dlmod`**' directory. The following line in the '**`/etc/SNMP/snmpd.conf`**' was appended so that the module was loaded on the start of the SNMP daemon:

```
dlmod nfdsSenseLink /usr/lib/snmp/dlmod/nfdsAgentPluginObject.so
```

4.4.3 ICMP Data Logging

The aim of this module is to collate the ICMP packets that are received on a Ethernet interface in order to build a table used later for packet aggregation by the application described later. A prototype was created for the first instance of development of the

agent for two reasons: to assess the suitability of libPCap for packet capturing [Jacobson] and to test the functionality without the complications of using SNMP in between.

LibPCap provides two methods for packet capturing: individual packet capturing and a packet capturing loop. The packet capturing loop, continuously loops to infinity until it is interrupted by a TERM signal from GNU/Linux, which would terminate the entire program as well. The individual-style however captures a single packet and processes it. The following is a detailed example of their use:

```
1. // LibPCap Initialisation
2.      dev_name = pcap_lookupdev(errbuf);
3.      pcap_lookupnet(dev_name, &dev_ipaddr, &dev_netmask, errbuf);
4.             handle = pcap_open_live(dev_name, BUFSIZ, 1, -1, errbuf);
5.      pcap_compile(handle, &filter, expression, 0, dev_netmask);
6.      pcap_setfilter(handle, &filter);
7.
8. // Individual Packet Capture
9.      packet = pcap_next(handle, &pkthdr);
10.     show_headers(NULL, NULL ,packet);
11.
12.// Looping Packet Capture
13.     pcap_loop(handle, -1, show_headers, NULL);
14.
15.// Close PCap Handle
16.     pcap_close(handle);
```

The first five lines of code shown, show the important process of setting up libPCap. The first line is a quick method of finding a compatible packet capture interface, which is useful for testing. The next line looks up the IP address and netmask for later use. The third line creates a handle for packet capture, putting the network device into promiscuous mode. The fourth and fifth line is an important phase; this is where it is possible to filter what packets can be captured, through the use of an expression string. In this example the expression string is set to “ICMP” to single out ICMP packets only, then the proceeding line applies this filter to the handle.

The first method seen after the initialisation captures once instance of a packet: its header contents and its payload. This packet can then be sent to a function for processing. The second variant shows the loop function, which technically does not return a value and should permanently block, so it takes a function name to pass the packets to. The last line closes the handle, releasing the promiscuous mode.

As the entire packet-capture process requires to be running all the time, the agent must be constructed in an entirely different was as opposed to a pluggable module, as it is technically impossible to capture packets and handle requests, as the packet capture blocks all request handling or vice-versa. Therefore a process must be created which attaches to the master agent using the AgentX protocol, so we can try to separate the two processes. This uses the net-SNMP libraries to do most the the work. Therefore we compile the plugin outside the main net-SNMP package and put a main() wrapper around it, making it not unlike a daemon.

```
1. snmp_enable_stderrlog();
2. if (agentx_subagent)
3. {
4.      netsnmp_ds_set_boolean(NETSNMP_DS_APPLICATION_ID, NETSNMP_DS_AGENT_ROLE, 1);
5. }
6.
7. init_agent("nfdsIcmpDaemon");
8. init_nfdsSniffIcmp();
9. init_snmp("nfdsIcmpDaemon");
10.
11./* If snmp master agent, initialise the ports */
12.if (!agentx_subagent)
13.     init_master_agent();
```

The above code is used in creating the basis of the main() wrapper, as all this does is initialises the agent and sets it up either as a master agent which is independent of the SNMP daemon, or as a sub-agent where it passes the data back to the SNMP daemon. The latter is the preferred choice, as further enhancements for the system in the future could already be partially implemented in the snmpd agent. Note however that the plugin itself is initialised at this stage (init_nfdsSniffIcmp()).

Once the daemon is created, another problem is evident: how is it possible to handle incoming requests whilst continually monitoring the connection? The answer is to fork a separate process for one of the tasks. As the SNMP side needs to be implemented and initialised first, the logical choice is to fork the packet capturing process. If we were to fork the request handler, it would be very hard to devise a method to inherit the initialised plugin values from init_nfdsSniffIcmp().

```
1. pipe(sock_pair);
2. if ( (pid = fork()) == 0)
3. {
4.      int type, code;
5.      struct in_addr *ipsrc;
6.      char *ptr;
7.
8.      close(sock_pair[0]);
9.      open_pcap();
10.     for ( ; ; )
11.     {
12.             ptr = next_pcap(&type, &code, ipsrc);
13.             message.type = type;
14.             message.code = code;
15.             message.in_addr = *ipsrc;
16.             write(sock_pair[1], &message, sizeof(icmp_t));
17.     }
18.     exit(0);
19.}
```

The above code sample shows a pipe being set-up for interprocess communication and the forking of the packet handling mechanism. The pipe serves as a way of serving incoming packets down a data stream to the original program thread, where the 'reading' end of the pipe is closed immediately. Open_pcap() initialises the packet capturing handle and then a continuous for loop is started, which calls the next_pcap function to get packets. The packets are then written into a user-defined structure (see Appendix for more information) and then they are sent down the pipe to the original program process.

```
1. char *next_pcap(int *type, int *code, struct in_addr *ipsrc)
2. {
3.      int ip_hlen;
4.      char *ptr;
5.      struct pcap_pkthdr hdr;
6.      struct ip *ip;
7.      struct icmp *icmp;
8.
9.      while ( (ptr = (char *) pcap_next(handle, &hdr)) == NULL);
10.
11.     ptr += 14;
12.     ip = (struct ip *) ptr;
13.     ip_hlen = ip->ip_hl << 2;
14.     icmp = (struct icmp *) (ptr + ip_hlen);
15.
16.     *type = icmp->icmp_type;
17.     *code = icmp->icmp_code;
18.     *ipsrc = ip->ip_dst;
19.
20.     return (ptr);
21.}
```

This is the capturing function that extracts the IP destination, ICMP type and ICMP code using various structure definitions to make the data extraction easy. Firstly the 'while' statement keeps looping to capture a packet that fits the filter information set in the open_pcap() function. The pcap_next function was used, as it proved too complex an issue to pass the end of the pipe to an external function via the pcap_loop function. The Ethernet header (14 bytes) is then bypassed, meaning we can split the remaining data into a IP header and a ICMP packet (calculated from the IP headers reported length). It is then possible to return the values.

```
1.      close(sock_pair[1]);
2.
3.      // Make reading NON-BLOCKING
4.      fcntl(sock_pair[0], F_SETFL, O_NONBLOCK);
5.      while(keep_running)
6.      {
7.              agent_check_and_process(0);
8.              if ( read(sock_pair[0], &message, sizeof(icmp_t)) > 0)
9.              {
10.                     log_icmp(message.type, message.code, message.in_addr);
11.                     num_received++;
12.             }
13.             usleep(50);
14.     }
15.
16.     snmp_shutdown("nfdsIcmpDaemon");
17.     return(0);
```

The above is the other side of the fork(), where the 'writing' side of the pipe is closed and set to non-blocking (O_NONBLOCK) and then a conditional while loop runs. Briefly to explain, this loop:

- Looks for requests and handles them, but in a non-blocking style (line 7).
- Logs packets as they received through the end of the pipe, so that they can be written to a SNMP table (lines 8-10).
- Increment a value indicating how many packets have been received (line 11).

If this loop is interrupted by user intervention (not implemented at time of writing), then line 16 closes the daemon and therefore the program and SNMP table. The code

```
1. void init_nfdsSniffIcmp(void)
2. {
3.         static oid nfdsIcmpStatsPacketsLogged_oid[] =
4.               { 1, 3, 6, 1, 4, 1, 8072, 2, 4, 1, 1, 2, 2, 1, 0 };
5.         static oid nfdsIcmpStatsPacketsBumped_oid[] =
6.               { 1, 3, 6, 1, 4, 1, 8072, 2, 4, 1, 1, 2, 2, 2, 0 };
7.         static oid nfdsIcmpConfigEntryLimit_oid[] =
8.               { 1, 3, 6, 1, 4, 1, 8072, 2, 4, 1, 1, 2, 1, 1, 0 };
9.         static oid nfdsIcmpConfigAgeOut_oid[] =
10.              { 1, 3, 6, 1, 4, 1, 8072, 2, 4, 1, 1, 2, 1, 2, 0 };
11.
12.        netsnmp_register_read_only_counter32_instance
13.                  ("nfdsIcmpStatsPacketsLogged",
14.                   nfdsIcmpStatsPacketsLogged_oid,
15.                   OID_LENGTH(nfdsIcmpStatsPacketsLogged_oid),
16.                   &num_received, NULL);
17.
18.        netsnmp_register_read_only_counter32_instance
19.                  ("nfdsIcmpStatsPacketsBumped",
20.                   nfdsIcmpStatsPacketsBumped_oid,
21.                   OID_LENGTH(nfdsIcmpStatsPacketsBumped_oid),
22.                   &num_deleted, NULL);
23.
24.        netsnmp_register_ulong_instance("nfdsIcmpConfigEntryLimit",
25.                                        nfdsIcmpConfigEntryLimit_oid,
26.                                        OID_LENGTH(nfdsIcmpConfigEntryLimit_oid),
27.                                        &max_logged,
28.                                        icmp_log_config_handler);
29.
30.        netsnmp_register_ulong_instance("nfdsIcmpConfigAgeOut",
31.                                        nfdsIcmpConfigAgeOut_oid,
32.                                        OID_LENGTH(nfdsIcmpConfigAgeOut_oid),
33.                                        &max_age,
34.                                        icmp_log_config_handler);
35.
36.        initialise_table_nfdsIcmpLogTable();
37.}
```

following this point now has a close resemblance to the media detection module.

This function initialises several variables and their OID's:

- **nfdsIcmpStatsPacketsLogged** – a static counter of the number of packets that have been logged since the agent has been running. Uses the num_received variable for its data.
- **nfdsIcmpStatsPacketsBumped** – a static counter of the number of packets that have been prematurely removed from the SNMP table. Uses the num_deleted variable for its data.
- **nfdsIcmpConfigEntryLimit** – a user-changeable setting for the maximum number of entries in the table to save memory on the host machine. Uses the max_logged variable and when modified, triggers the a log configuration handler (line 28).
- **nfdsIcmpConfigAgeOut** – a user-changeable setting for setting the maximum length of time a packet can exist within the table before it is deleted. Uses the max_age variable and when modified, triggers a log configuration handler (line 34).

Finally, to conserve space, a separate function is called to create the SNMP table:

```
1. void initialise_table_nfdsIcmpLogTable(void)
2. {
3.     static oid nfdsIcmpLogTable_oid[] =
4.            { 1, 3, 6, 1, 4, 1, 8072, 2, 4, 1, 1, 2, 3, 1 };
5.     size_t nfdsIcmpLogTable_oid_len = OID_LENGTH(nfdsIcmpLogTable_oid);
6.
7.     nfdsIcmpLogTable = netsnmp_create_table_data_set("nfdsIcmpLogTable");
8.     netsnmp_table_dataset_add_index(nfdsIcmpLogTable, ASN_UNSIGNED);
9.
10.    netsnmp_table_set_add_default_row(nfdsIcmpLogTable,
11.                                      COLUMN_NFDSICMPLOGTIME,
12.                                      ASN_TIMETICKS, 0, NULL, 0);
13.    netsnmp_table_set_add_default_row(nfdsIcmpLogTable,
14.                                      COLUMN_NFDSICMPLOGDATEANDTIME,
15.                                      ASN_OCTET_STR, 0, NULL, 0);
16.    netsnmp_table_set_add_default_row(nfdsIcmpLogTable,
17.                                      COLUMN_NFDSICMPLOGTYPE,
18.                                      ASN_INTEGER, 0, NULL, 0);
19.    netsnmp_table_set_add_default_row(nfdsIcmpLogTable,
20.                                      COLUMN_NFDSICMPLOGCODE,
21.                                      ASN_INTEGER, 0, NULL, 0);
22.    netsnmp_table_set_add_default_row(nfdsIcmpLogTable,
23.                                      COLUMN_NFDSICMPLOGIPSRC,
24.                                      ASN_IPADDRESS, 0, NULL, 0);
25.
26.    netsnmp_register_table_data_set(netsnmp_create_handler_registration(
27.                                    "nfdsIcmpLogTable",
28.                                    nfdsIcmpLogTable_handler,
29.                                    nfdsIcmpLogTable_oid,
30.                                    nfdsIcmpLogTable_oid_len,
31.                                    HANDLER_CAN_RWRITE), nfdsIcmpLogTable, NULL);
32.
33.    // Check log size/age every 30 secs
34.    snmp_alarm_register(30, SA_REPEAT, check_log_size, NULL);
35.}
```

This function creates the table, OID and handler registrations. At first the table instance is created (line 7) and then indexes are added for performance (line 8). We then add 5 columns: Log-time in “ticks”, date and time as a string, ICMP type as an integer, ICMP code as an integer and finally the IP source as a four byte value. The last two processes is to register a table handler (unused at the time of writing, as all data should be valid) and a handler to check the size of the table every 30 seconds (to flush out old entries).

This is the piece of code that maintains the table. It is used to remove old entries that are older than the value set in the nfdsIcmpConfigAgeOut (max_age) SNMP variable. When there are more entries than specified in nfdsIcmpConfigEntryLimit (max_logged), it shall remove the oldest ones to make room and conserve space.

```
1. void check_log_size(unsigned int clientreg, void *clientarg)
2. {
3.         netsnmp_table_row *row, *deleterow, *tmprow;
4.         netsnmp_table_data_set_storage *data;
5.         u_long          count = 0;
6.         struct timeval  now;
7.         long            uptime;
8.
9.         gettimeofday(&now, NULL);
10.        uptime = netsnmp_timeval_uptime(&now);
11.
12.        for (row = nfdsIcmpLogTable->table->first_row; row ; row = row->next)
13.        {
14.                count++;
15.                if (max_logged && count == max_logged)
16.                        break;
17.
18.                data = (netsnmp_table_data_set_storage *) row->data;
19.                data = netsnmp_table_data_set_find_column(data,
   COLUMN_NFDSICMPLOGTIME);
20.             if (max_age &&
21.                uptime > (*(data->data.integer) + max_age * 100 * 60))
22.                   netsnmp_table_dataset_remove_and_delete_row(
23.                                                    nfdsIcmpLogTable,
24.                                                           row);
25.     }
26.
27.     if (!row)
28.             return;
29.
30.    for (deleterow = nfdsIcmpLogTable->table->first_row, row = row->next;
31.             row; row = row->next)
32.     {
33.                tmprow = deleterow->next;
34.                netsnmp_table_dataset_remove_and_delete_row(nfdsIcmpLogTable,
35.                                                            deleterow);
36.                deleterow = tmprow;
37.                num_deleted++;
38.         }
39.}
```

To eliminate old entries, the time is taken and is converted to “ticks” (line 9-10). Once the time is logged, the function goes through the entire table with a “for” loop (line 12) and compares the time each row was logged (18-21) with the current time to see if it has expired compared to max_age. If this is the case, the row is deleted accordingly (line 22). Inside the “for” loop another check is done to see if there are too many entries present compared to max_logged(line 14-16), if this is indeed the case, the loop exits early. The “if” statement at line 27-28 checks to see if the loop has indeed exited early, by checking to see if row has been defined (the last row would have had a pointer to a null value, thus setting “row” to null on the last pass.). If this row is set to null, then the program can exit the function, otherwise the program has determined there is too many entries. Carrying on from the last row the program was at, the for loop continues (line 30-31), removing each of the excess rows (33-36), as the rows near the end are the oldest. To alert the user that there has been a premature removal of rows, the value nfdsIcmpStatsPacketsBumped (num_deleted) is incremented (line 37).

The final function is the one that's called by the main() body of the program to log the ICMP type/code and IP source from the other side of the pipe. The function creates a blank data row (line 14) and inserts:

```
1. void log_icmp(int type, int code, struct in_addr ipsrc)
2. {
3.      long           uptime;
4.      struct timeval        now;
5.      netsnmp_table_row *row;
6.      u_char         *logdate;
7.      size_t          logdate_size;
8.      time_t          timetnow;
9.
10.     if (netsnmp_ds_get_boolean(NETSNMP_DS_APPLICATION_ID,
11.                                NETSNMP_DS_APP_DONT_LOG))
12.             return;
13.
14.     row = netsnmp_create_table_data_row();
15.
16.     default_num++;
17.     netsnmp_table_row_add_index(row, ASN_UNSIGNED, &default_num,
18.                                 sizeof(default_num));
19.
20.     gettimeofday(&now, NULL);
21.     uptime = netsnmp_timeval_uptime(&now);
22.     netsnmp_set_row_column(row,
23.                            COLUMN_NFDSICMPLOGTIME,
24.                            ASN_TIMETICKS,
25.                             (u_char *) & uptime,
26.                            sizeof(uptime));
27.     time(&timetnow);
28.     logdate = date_n_time(&timetnow, &logdate_size);
29.     netsnmp_set_row_column(row,
30.                            COLUMN_NFDSICMPLOGDATEANDTIME,
31.                            ASN_OCTET_STR,
32.                             logdate,
33.                            logdate_size);
34.     netsnmp_set_row_column(row,
35.                            COLUMN_NFDSICMPLOGTYPE,
36.                            ASN_INTEGER,
37.                            (u_char *) & type, sizeof(type));
38.     netsnmp_set_row_column(row,
39.                                 COLUMN_NFDSICMPLOGCODE,
40.                                 ASN_INTEGER,
41.                                 (u_char *) & code, sizeof(code));
42.     netsnmp_set_row_column(row,
43.                                 COLUMN_NFDSICMPLOGIPSRC,
44.                                 ASN_IPADDRESS,
45.                                 (u_char *) & ipsrc, sizeof(ipsrc));
46.
47.     netsnmp_table_dataset_add_row(nfdsIcmpLogTable, row);
48.     check_log_size(0, NULL);
49.}
```

- An index. (line 16-18)
- Time in “ticks” from the current, therefore “log” time. (line 20-26)
- Log time in string format. (line 27-33)
- ICMP type to log. (line 34-37)
- ICMP code to log. (line 38-41)
- IP source to log. (line 42-45)

Finally, the row can be added (line 47) and a check called to see if the newly added row has made the table exceed is maximum number of entries (line 48).

To incorporate the table into the MIB, the following the highlights of what should be entered for the table (for the whole listing including settings/counters, see Appendix):

The main point to note is that the entire table consists of a sequence of log entries

```
1. nfdsIcmpLogTable OBJECT-TYPE
2.      SYNTAX                SEQUENCE OF NfdsIcmpLogEntry
3.      MAX-ACCESS     not-accessible
4.      STATUS         current
5.      DESCRIPTION
6.      "A table of packet log entries."
7.      ::= { nfdsIcmpLog 1 }
8.
9. nfdsIcmpLogEntry OBJECT-TYPE
10.     SYNTAX                NfdsIcmpLogEntry
11.     MAX-ACCESS      not-accessible
12.         STATUS          current
13.         DESCRIPTION
14.     "A packet log entry."
15.     INDEX   { nfdsIcmpLogIndex }
16.     ::= { nfdsIcmpLogTable 1 }
17.
18.NfdsIcmpLogEntry ::= SEQUENCE {
19.     nfdsIcmpLogIndex      Unsigned32,
20.     nfdsIcmpLogTime               TimeStamp,
21.     nfdsIcmpLogDateAndTime        DateAndTime,
22.     nfdsIcmpLogType               Integer32,
23.     nfdsIcmpLogCode               Integer32,
24.     nfdsIcmpLogIpSrc      IpAddress
25.}
26.
27.nfdsIcmpLogIndex OBJECT-TYPE
28.     SYNTAX Unsigned32
29.     MAX-ACCESS not-accessible
30.     STATUS current
31.     DESCRIPTION
32.             "The index record of the log."
33.     ::= { nfdsIcmpLogEntry 1 }
34.
35.-- <SNIP>
36.
37.nfdsIcmpLogIpSrc OBJECT-TYPE
38.     SYNTAX IpAddress
39.     MAX-ACCESS read-only
40.     STATUS current
41.     DESCRIPTION
42.     "The IP Address for the source of the incoming ICMP packet."
43.     ::= { nfdsIcmpLogEntry 6 }
```

(line 10, 18), where each entry is a collection of several object types.

Compilation of the daemon is straightforward like the plugin, as a makefile has been created to automate the compilation procedure. When compiled a binary called nfdsIcmpDaemon is created, which can be run from any directory, as long as access is available for AgentX in the snmpd.conf:

```
master agentx
```

4.5 Creation of the Application

The application acts as a management station for the network fault diagnosis system, as it brings together all the host data into one location and therefore allows the user to visually see problems through a GUI.

4.5.1 SNMP Data Retrieval

The application must be able to collate two types of information: the network media status and the logged ICMP packets upon each host. The retrieval of the media status

information is achieved by issuing a SNMP "GET" request to each host, therefore receiving only one variable as a result, which should be a integer representing the status of the network.

```
1.      for (hs = sessions, hp = nic_hosts; hp->name; hs++, hp++)
2.      {
3.              struct snmp_pdu *req;
4.              struct snmp_session sess;
5.              struct variable_list *vars;
6.              snmp_sess_init(&sess);
7.
8.              sess.version = SNMP_VERSION_1;
9.              fprintf(stderr, "NIC: %s\n", hp->name);
10.             sess.peername = strdup(hp->name);
11.             sess.community = strdup(hp->community);
12.             sess.community_len = strlen(sess.community);
13.             init_snmp("nic_poll");
14.             OidLen = sizeof(Oid)/sizeof(Oid[0]);
15.             if (!(hs->sess = snmp_open(&sess)))
16.             {
17.                     snmp_perror("snmp_open");
18.                     continue;
19.             }
20.             if (!read_objid(Oid_name, Oid, &OidLen))
21.             {
22.                     fprintf(stderr, "OID name resolution failed");
23.             }
24.             req = snmp_pdu_create(SNMP_MSG_GET);    /* send the  GET */
25.             snmp_add_null_var(req, Oid, OidLen);
26.
27.             status = snmp_synch_response(hs->sess, req, &response);
28.             if (status == STAT_SUCCESS && response->errstat == SNMP_ERR_NOERROR) {
29.                             for(vars = response->variables; vars; vars = vars-
   >next_variable) {
30.                             if (vars->type == ASN_INTEGER)
31.                                     nic_list_insert(&(*list), *(vars-
   >val.integer), sess.peername);
32.                             else
33.                                     fprintf(stderr, "value is NOT a long!\n");
34.                             }
35.             } else {
36.                             if (status == STAT_SUCCESS)
37.                             fprintf(stderr, "Error in packet\nReason: %s\n",
38.                                     snmp_errstring(response->errstat));
39.                             else
40.                             snmp_sess_perror("snmpget", hs->sess);
41.             }
42.             if (response)
43.                             snmp_free_pdu(response);
44.             snmp_close(hs->sess);
45.     }
```

To access multiple hosts and read their media status values, the above function is used. The 'for' loop uses two structure instances that are user defined (see appendix) so that session and hosts can be processed in turn. For each occurrence of the loop, a SNMP session is created, with the SNMP version, hostname (peername) and community (all declared in lines 8-12). After initialising the SNMP session, line 20-23 show how 'read_objid' is used to resolve an Object ID Name (e.g. nfdsSenseLink.0) to a dot-separated Object ID for SNMP usage. Checking that the SNMP session is open for use, it should be possible to create a blank PDU (line 24), set the OID to connect to (line 25) and send the PDU off (line 27). When this has been done, the system waits for a response, waiting for a reply in a synchronous manner. To implement this in an asynchronous manner would be complex to implement in a callback function where parameter passing is limited.

The 'if' statement on line 28 checks to ensure that the response is valid before entering a 'for' loop. As the agent only handles one interface (at the time of writing), the 'for' loop only completes one pass. This pass verifies that the data is of integer type ('long'-type to be exact) and if this is the case, then the integer and the hostname of the current session is inserted into a customised array (linked-list).

The collation of ICMP table data via SNMP is a different affair, as the collation of data is very complicated. To eliminate the complexity, the source of the program 'snmptable' was perused for possible adaptation, which is part of the Net-SNMP package. This program provides different ways of collecting, interpreting and displaying data contained within SNMP tables, so the code was copied and modified heavily, so that all unnecessary code was removed and the original code optimised to return the results so that they could be inserted into a linked-list with ease.

```
1.      get_node(tblname, root, &rootlen);
2.      if (get_fields(tblname) == icmp_fields)
3.      {
4.              SOCK_STARTUP;
5.              if (!(hs->sess = snmp_open(&sess))) {
6.                      snmp_perror("snmp_open");
7.                      continue;
8.              }
9.              entries_found = 0;
10.             errors = get_entries(hs->sess);
11.             snmp_close(hs->sess);
12.             SOCK_CLEANUP;
13.             if (!errors)
14.             {
15.                     dp = data;
16.                     for (entry = 0; entry < entries_found; entry++)
17.                     {
18.                             icmp_list_insert(&(*list), dp[4], dp[3], atoi(dp[2]),
   atoi(dp[1]), dp[0]);
19.                             dp += icmp_fields;
20.                     }
21.                     continue;
22.             }
23.             else
24.                     continue;
25.     }
```

Although the ICMP data collection uses the same 'for' loop and sessions initiation (not shown), the operation on tables is slightly different, as instead of returning rows, the SNMP protocol returns the contents of each column in turn, which makes array inserts ion complex. Lines one and two depict how the table name is resolved then the number of fields are counted and checked with an modified version of the get_fields function (AKA get_field_names, but reduced from 117 lines of code to 46 for optimisation purposes) from Net-SNMP's 'snmptable' program. If the number of entries is correct, the session is opened, then yet another modified function named get_entries (aka get_table_entries, but reduced from 218 lines to 96 for optimisation purposes) is used from 'snmptable' to gather the data into a global data array called 'data'. This report shall refrain from discussing the techniques used to pull the data from the SNMP table, as it closely resembles the NIC SNMP data collection but with some complex data manipulation using tedious methods such as string and pointer manipulation, memory allocation, etc.

The drawback from using these functions is that the data retrieved is in byte-format and reverse order. Therefore for each data entry found by get_entries, the ICMP data for a specific entry is inserted in reverse order into the array (line 18), with some data conversion (ASCII data to integer for example).

```
1. void icmp_list_insert(struct icmp_entry **list,
2.                       char *host, char *date, int type, int code, char *ipsrc)
3. {
4.         struct icmp_entry *next;
5.         if(*list == NULL)
6.         {
7.                 next = *list;
8.                 *list = malloc(sizeof(struct icmp_entry));
9.                 (*list)->host = host;
10.                (*list)->date = date;
11.                (*list)->type = type;
12.                (*list)->code = code;
13.                (*list)->ipsrc = ipsrc;
14.                (*list)->next = next;
15.                return;
16.        }
17.        icmp_list_insert(&((*list)->next), host, date, type, code, ipsrc);
18.}
```

This is a short example of how the linked-list insertion process works, in this case with the ICMP array. The NIC array is only structurally different. Data passed to the function is allocated a space at the end of the array as the function continues to call itself recursively (line 17) until the end is reached (line 6). When the end is reached, the new node allocation takes place (line 8) and the data is entered into a custom storage structure (line 9-13), the new entry is added (line 14) and the recursive cycle ends with a 'return' (line 15).

```
1. int icmp_list_free(struct icmp_entry **list)
2. {
3.      struct icmp_entry *next;
4.
5.      while (*list != NULL)
6.      {
7.              next = (*list)->next;
8.              free(*list);
9.              *list = next;
10.     }
11.     return 0;
12.}
```

Finally, this code represents how the linked-list is cleansed before a refresh. The 'while' loop (line 5) cycles through each node in the list, freeing each node from memory (line 8).

4.5.2 Graphical Data Representation

For the construction of the application, it was decided that the Gimp (GNU Image Manipulation Program) Toolkit, otherwise known as GTK, would be the best choice for window/widget creation, as it is used in most GNU/Linux application front-ends. This would also have the added bonus of working on other platforms as well, as the GTK toolkit is available for propriatory operating systems such as Microsoft Windows, making the portability of the application a distinct possibility in future.

Another reason for using GTK was the ability to render table structures in the GUI effectively. It was decided that this was the best choice for displaying lists of ICMP packets and hosts media status, as it could allow the sorting and clear representation of data for the user to interpret. The GUI interface was therefore split into vertical sections: ICMP table and host NIC status.

```
1.      app_window = make_app_window();
2.
3.      hpaned = gtk_hpaned_new();
4.      gtk_container_add (GTK_CONTAINER(app_window), hpaned);
5.      gtk_widget_show(hpaned);
6.
7.      icmp_model = make_icmp_model();
8.      make_icmp_table_view(icmp_model, &icmp_align, &icmp_tree);
9.      gtk_paned_add1(GTK_PANED(hpaned), icmp_align);
10.     gtk_widget_show(icmp_align);
11.
12.     nic_model = make_nic_model();
13.     make_nic_table_view(nic_model, &nic_align, &nic_tree);
14.     gtk_paned_add2(GTK_PANED(hpaned), nic_align);
15.     gtk_widget_show(nic_align);
16.
17.     gtk_widget_show_all(app_window);
18.
19.     icmp_update_callback_tag = g_timeout_add(5000,
20.                                icmp_update_callback,
21.                                icmp_tree);
22.
23.     nic_update_callback_tag = g_timeout_add(30000,
24.                           nic_update_callback,
25.                           nic_tree);
```

The above code is from the make_application function and forms the main body of the program. The first line calls a trivial function to make the main window outline for the two tables, where the containers for these are designated in lines 3-5. These lines tell the application that any objects inserted into the window shall be inserted from left to right along the horizontal axis. The code from here on does the following:

1. Creates a model for the data to conform to (lines 7,12)
2. Maps the data model onto a customisable GUI table (lines 8, 13)
3. Adds the table to the GUI into a pane and displays it (lines 9, 10, 14, 15, 17)
4. Creates a callback function to update the data in the tables (19-25)

```
static GtkTreeModel *make_icmp_model(void)
{
       GtkListStore   *icmp_model;

       icmp_model = gtk_list_store_new(ICMP_NUM_COLUMNS,
                                       G_TYPE_STRING,
                                       G_TYPE_STRING,
                                       G_TYPE_INT,
                                       G_TYPE_INT,
                                       G_TYPE_STRING,
                                       GDK_TYPE_COLOR);
       return GTK_TREE_MODEL(icmp_model);
}
```

This is an example of how a data model is defined, taken from the ICMP table model. The model is essentially a definition of the number of columns and the data types associated with each column in particular. Once this has been defined, the model is

then create as a list data store.

```
1.      renderer = gtk_cell_renderer_text_new();
2.      nic_tree = gtk_tree_view_new_with_model(nic_model);
3.      gtk_object_set(GTK_OBJECT(nic_tree), "headers-hidden", TRUE, "rules-hint",
   TRUE, NULL);
4.
5.      selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(nic_tree));
6.      gtk_tree_selection_set_mode(selection, GTK_SELECTION_NONE);
7.
8.      column = gtk_tree_view_column_new_with_attributes("#",
9.                                                        renderer,
10.                                                       "text",
11.                                                       NIC_COLUMN_STAT,
12.                                                       "background-gdk",
13.                                                       NIC_COLUMN_COLOUR,
14.                                                       NULL);
15.     gtk_tree_view_column_set_resizable(column, TRUE);
16.     gtk_tree_view_append_column(GTK_TREE_VIEW(nic_tree), column);
17.     gtk_tree_view_column_set_sort_column_id(column, NIC_COLUMN_STAT);
18.
19.     column = gtk_tree_view_column_new_with_attributes("Hostname",
20.                                                       renderer,
21.                                                       "text",
22.                                                       NIC_COLUMN_HOST,
23.                                                       NULL);
24.     gtk_tree_view_column_set_resizable(column, TRUE);
25.     gtk_tree_view_append_column(GTK_TREE_VIEW(nic_tree), column);
26.     gtk_tree_view_column_set_sort_column_id(column, NIC_COLUMN_HOST);
27.
28.     nic_scroll = gtk_scrolled_window_new(NULL, NULL);
29.     gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(nic_scroll),
30.                                    GTK_POLICY_NEVER,
31.                                    GTK_POLICY_AUTOMATIC);
32.     gtk_container_add(GTK_CONTAINER(nic_scroll), nic_tree);
33.
34.     align = gtk_alignment_new(0, 0, 1, 1);
35.     gtk_container_add(GTK_CONTAINER(align), nic_scroll);
36.
37.     *align_return = align;
38.     *tree_return = nic_tree;
```

This is the source code for the creation of the make_*_table_view which creates the GUI table for the data to be represented within. Line number two is of special note, as this line draws in the required data model which maps on to the table in line three, but with added options regarding the display of each particular column. Lines 8 to 17 show how a single column is added. The first seven lines of this particular block of code, show how the column header tab is set, what format the entries take and finally what other properties have to be set, such as colour in this example. Lines 15 to17 then show how the column is added and how it should behave with resizing and sorting (when the tab at the top is clicked). The same process is repeated for lines 19 to 26. The lines from 28 onwards show how the table scrolls (28-32) and the addition of the table to the main application (34-38).

After the addition of both tables, the program looks like the following:

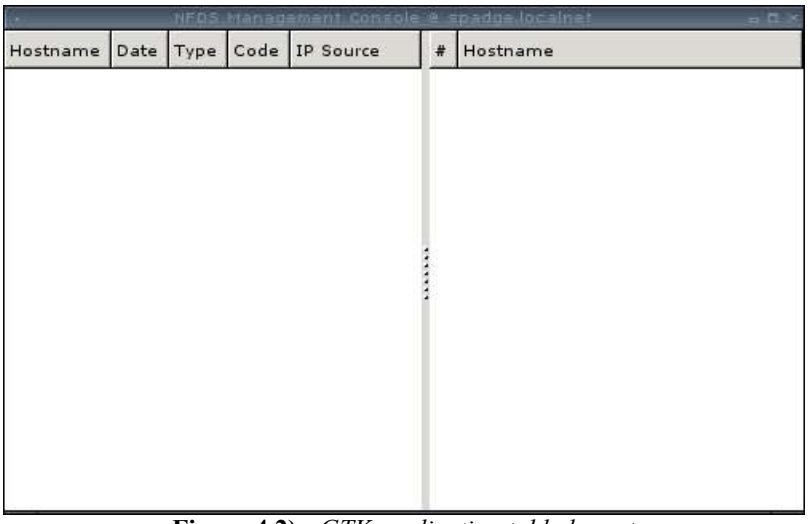


Figure 4.2) - *GTK application table layout*

The application however is useless at its present state, as the main body just draws a window and there is typically no linear process path within a graphical application. Therefore either user intervention or a callback function should be used to initiate actions on the application. At the time of writing, a basic callback function is all that is required.

This callback function is used to populate the ICMP table. Line 9 to 17 deals with initialising and calibrating colour values should they be needed to highlight particular columns. Line 19 to 21 is a piece of code that is used to detect if the current window is in the foreground, therefore if it is in the background it exits at this point. This function means that bandwidth can be saved when the application is not in use. Thereafter (23 onwards), the function now deals with collecting data (line 24 call of icmp_list_make – see section 4.5.1) and then creating and initialising the GTK list store, which is linked into the main GUI table. Then, looping through the linked list nodes (29-41), the contents of each node is copied to the appropriate list store columns, therefore instantaneously updating the table contents in front of the users eyes. Once this process is complete, the original linked list is freed before the next instance of the callback is created.

```
1. static gint icmp_update_callback (gpointer view_void)
2. {
3.      GtkWidget *view;
4.      GtkListStore *store;
5.      struct icmp_entry *ml, *il;
6.      GdkColor              colour[1];
7.      gboolean       success[1];
8.
9.      colour[0].red = colour[0].green = colour[0].blue = 0xffff;
10.     gdk_colormap_alloc_colors(
11.             gdk_colormap_get_system(),
12.             colour,
13.             1,
14.             FALSE,
15.             FALSE,
16.             success
17.     );
18.
19.     view = GTK_WIDGET(view_void);
20.     if (!manage_is_active(view))
21.             return TRUE;
22.
23.     ml = NULL;
24.     if (icmp_list_make(&ml))
25.             return TRUE;
26.
27.     store = GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(view)));
28.     gtk_list_store_clear(store);
29.     for (il = ml; il; il = il->next)
30.     {
31.             GtkTreeIter iter;
32.             gtk_list_store_append(store, &iter);
33.             gtk_list_store_set(store, &iter,
34.                     ICMP_COLUMN_HOST, il->host,
35.                   ICMP_COLUMN_DATE, il->date,
36.                   ICMP_COLUMN_TYPE, il->type,
37.                   ICMP_COLUMN_CODE, il->code,
38.                   ICMP_COLUMN_IPSRC, il->ipsrc,
39.                   ICMP_COLUMN_COLOUR, colour,
40.                        -1);
41.     }
42.     icmp_list_free(&ml);
43.             return TRUE;
44.}
```

4.5.3 Packet Aggregation Model Integration

As the packets were collated to a single array, this array had to be processed to determine which model brought back the biggest aggregate. After this was determined, the aggregate model that was dominant was returned and the details of the packet returned so the entries that fit the model and the appropriate details could be highlighted.

The above function, 'agg_insert', fills a specified array with the type and number of packets that conform to the aggregate value provided (the variable 'mod'). The function recursively calls itself (line 32), until it reaches the end of a specified array (line 5). If the end of the array is reached, then the packet is entered into the array as a unique packet (lines 7-15). Line 17 to 27 is a large 'if' statement that determines if a packet that conforms to the aggregate has already been entered. Line 18 is a generic match, if the packet cannot match the type or the code, then it must not exist already. If the packet matches a similar packet in accordance to a packet aggregation model, then the packet is not re-entered, but the one that it matches in incremented. Here are the three model examples directly linked to the code:

```
1. void
2. agg_insert(struct agg_entry **list, char *host, int type, int code, char *ipsrc,
   int mod)
3. {
4.      struct agg_entry *next;
5.      if (*list == NULL)
6.      {
7.              next = *list;
8.              *list = malloc(sizeof(struct agg_entry));
9.              (*list)->host = host;
10.             (*list)->type = type;
11.             (*list)->code = code;
12.             (*list)->ipsrc = ipsrc;
13.             (*list)->count = 1;
14.             (*list)->next = next;
15.             return;
16.     }
17.     else if (
18.              (((*list)->type == type) && ((*list)->code == code))
19.               &&
20.              (
21.               (((*list)->host == host) && ((*list)->ipsrc == ipsrc) && (mod = 0))
22.                ||
23.               (((*list)->ipsrc == ipsrc) && (mod = 1))
24.                ||
25.               (((*list)->host == host) && (mod = 2))
26.              )
27.             )
28.     {
29.             (*list)->count++;
30.             return;
31.     }
32.     agg_insert(&((*list)->next), host, type, code, ipsrc, mod);
33.}
```

- Line 21 - if the host IP and the IP source match, then this is a potential 1:1 candidate.
- Line 23 – if the IP source matches, this is a potential 1:N candidate.
- Line 24 – if the host IP matches, this is a potential N:1 candidate.

```
1.      for (temp = *list; temp; temp = temp->next)
2.      {
3.              agg_insert(&oto, temp->host, temp->type, temp->code, temp->ipsrc, 0);
4.              agg_insert(&otn, temp->host, temp->type, temp->code, temp->ipsrc, 1);
5.              agg_insert(&nto, temp->host, temp->type, temp->code, temp->ipsrc, 2);
6.      }
7.      agg = NULL;
8.      for ( agg = oto; agg; agg = agg->next ) {
9.              if (agg->count > max_count) {
10.                     *(host) = agg->host;
11.                     *(type) = agg->type;
12.                     *(code) = agg->code;
13.                     *(ipsrc) = agg->ipsrc;
14.                     agg_value = 0;
15.             }
16.     }
17.      <SNIP>
```

This is an excerpt from the packet aggregate function 'agg_packet'. This uses the previous function in lines three to five to populate three arrays for each of the three models, from the collated SNMP agent ICMP tables. Once this is has been done, each of the arrays will contain a count of all the possible candidates. Lines 8 to 16 shows how one of these arrays are then duly processed extracting the details if the highest value is found. This is then repeated for the remaining two arrays and returns the:

- aggregate model that was matched the most
- the details of the packet

Finally this information is then used to highlight the most popular model-type found within the whole array:

```
1.              if ((il->type == type) && (il->code == code)) {
2.                      if ((agg == 0) &&
3.                           (il->ipsrc == ipsrc) && (il->host == host))
4.                             colour[0].blue = colour[0].green = 0x9999;
5.                      if ((agg == 1) && (il->ipsrc == ipsrc))
6.                             colour[0].red = colour[0].green = 0x9999;
7.                      if ((agg == 2) && (il->host == host))
8.                             colour[0].blue = colour[0].red = 0x9999;
9.                      else
10.                            colour[0].blue = colour[0].green = colour[0].red=
   0xffff;
11.             }
12.             else
13.                     colour[0].red = 0xffff;
14.             gtk_list_store_append(store, &iter);
15.             gtk_list_store_set(store, &iter,
16.                          ICMP_COLUMN_HOST, il->host,
17.                     ICMP_COLUMN_DATE, il->date,
18.                     ICMP_COLUMN_TYPE, il->type,
19.                     ICMP_COLUMN_CODE, il->code,
20.                     ICMP_COLUMN_IPSRC, il->ipsrc,
21.                     ICMP_COLUMN_COLOUR, colour,          // This is the colour
22.                          -1);
```

As seen in a similar 'if' statement in the previous code, the 'if' statements check if the current entry matches the largest aggregate, if this is true, then the colour of the row is changed by the colour variable, inserted in line 21.

4.6 Conclusions

What follows is a screen-shot of the resultant application utilising both agents (ICMP collection and media detection of three instances all working on the local machine. The information collected is the result of polling the agents, which shall be tested thoroughly in the proceeding chapter.

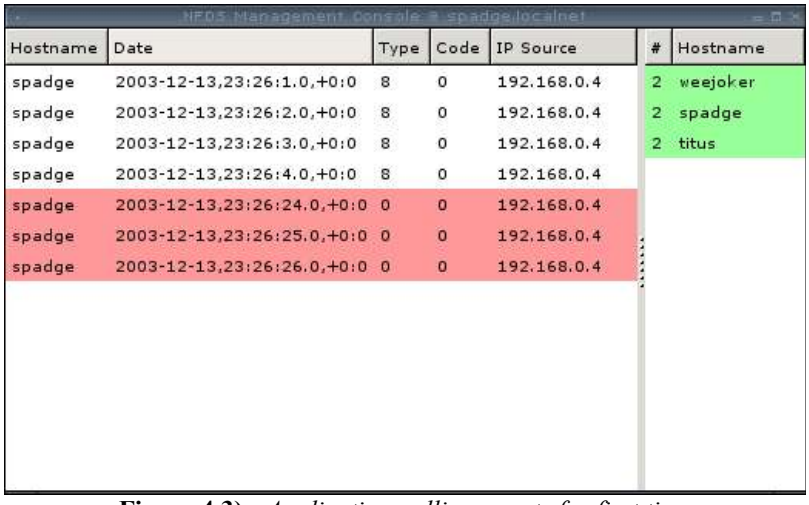


Figure 4.3) - *Application polling agents for first time*

5 Testing and Analysis

5.1 Introduction

The purpose of this section is to review the working application, agents, aggregation [Kanamaru (2000)] process and to test the functionality and their effectiveness of detecting faults. The code provided in the previous section was used in the core functionality of the agents and application, and this code shall be verified to see if the data retrieved and displayed is representative of the network status. The packet aggregation techniques and network media detection representation functions embedded in the code of the application, shall be tested to see if they can highlight faults upon the network in a suitable manner.

5.2 Network Media Detection

The purpose of testing the network media detection is perhaps the most important test phases, as the separate values returned by the agents should be able to detect network-card and media related faults with 100% certainty of diagnosis.

At the time of writing, only one particular network interface on a host could be polled, so therefore it would be logical to monitor an outward facing interface. This meant that on test machines, it would be preferable to set this interface to 'eth0', the first logical Ethernet interface. Here is a diagram of how this was implemented in tests on a test network:

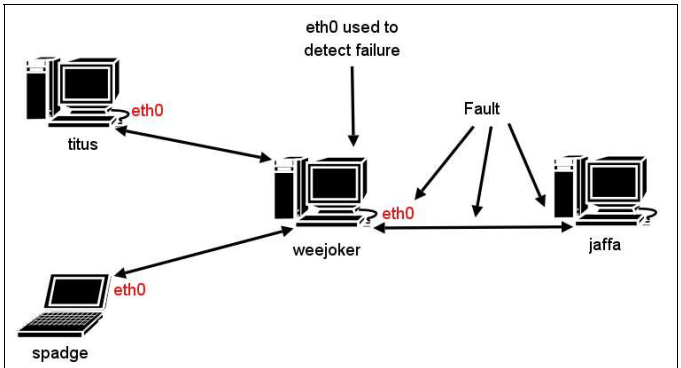


Figure 5.1) - *Media test scenario*

If the management station is located on 'spadge' as seen in the diagram, there are two nodes where potential and accurate failure detection can take place: the outgoing Ethernet interfaces of 'eth0' on 'spadge' and 'weejoker'. There is a potential failure scenario where it is impossible to contact both 'jaffa' and 'weejoker', as the interfaces on these face inwards towards the management station. Thankfully, if no node details are found the node is not highlighted by either green, red or yellow, but by grey instead.

As the management station was installed on 'spadge', the agents were therefore installed on both 'titus', 'weejoker' and 'spadge'. The machine named 'jaffa' was excluded due to time constraints, so was therefore implemented as a “problem” machine. The following is a screen shot of the set-up fully working:

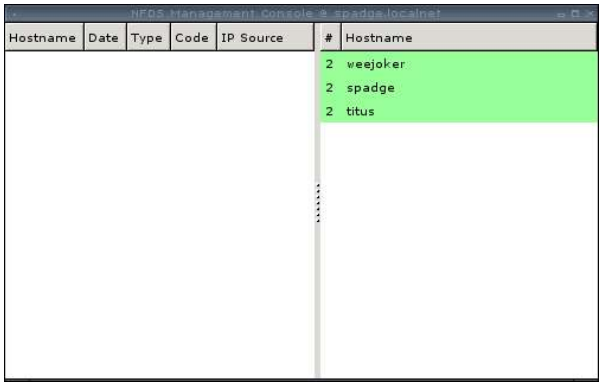


Figure 5.2) - *All nodes working with NIC agent*

During the testing of the software, it was noticed that the ICMP table log MIB entries were interfering with the NIC MIB operation when a NIC was faulty. After commenting out the faulty part of the MIB, it was possible to get the set-up to work fully, displaying red when the Linux NIC modules were removed, yellow when a cable was pulled and grey when the host was unreachable, although a great amount of application slowdown was observed.

A good example of the functionality working well is when the cable is pulled out of the NIC socket at the management station ('spadge'):

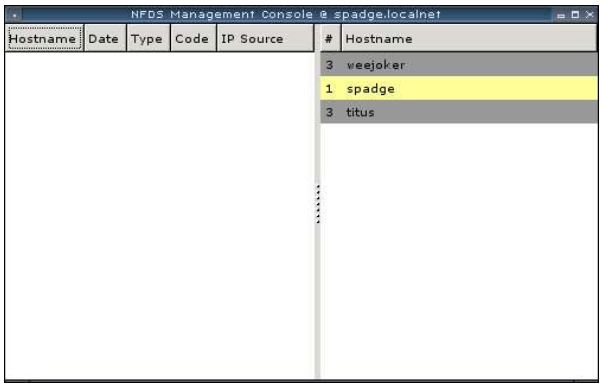


Figure 5.3)- *Screenshot of cable fault*

5.3 1:1 Packet Aggregation

The one-to-one relationship is affiliated where a client and a server have a specific, unique problem between themselves, but the problem does not affect other machines unless the problem causes the degradation of any services located upon the client or the host. It is therefore possible to characterise this behaviour as being either a problematic node or connection. This test scenario emulates a problem where a firewall blocks certain client to server traffic, thus making a single connection unresponsive, yet allowing others to work as normal:

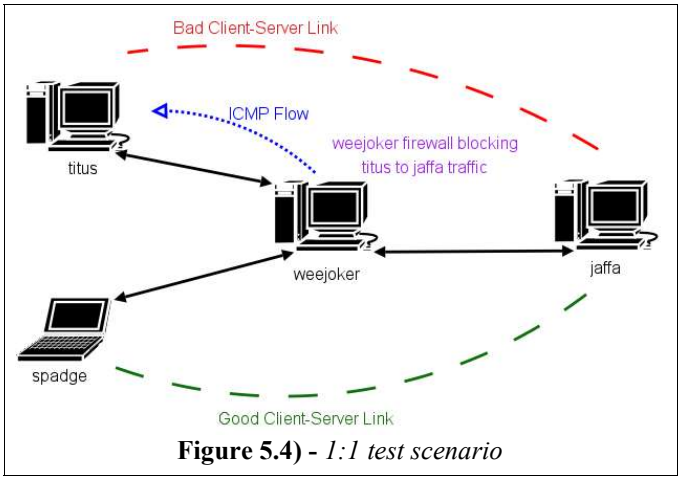
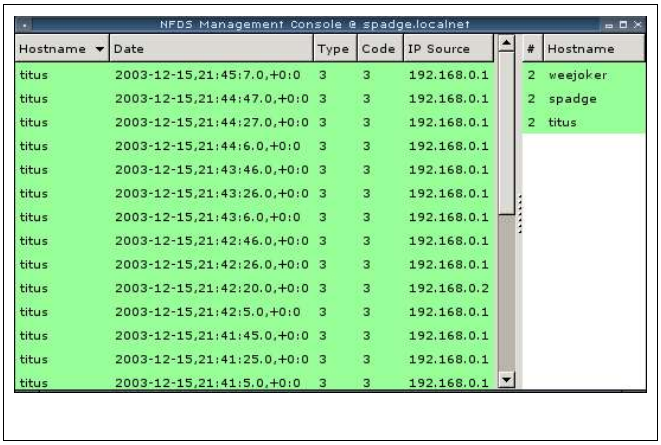


Figure 5.4) - *1:1 test scenario*

To emulate this scenario, a block was inserted on the 'weejoker' machine using the IP Tables firewall integrated with the Linux kernel. This was expected to return ICMP packets to the 'titus' machine, but not the machine named 'spadge'.

The following screen shot was observed:



This visual shows that the 'titus' machine was clearly affected when it tried to contact 'jaffa', resulting in the receipt of a multitude of ICMP port-unreachable packets at 'titus'. This model therefore can be assumed to work correctly. The problem could not be detected however if the server was tightly secured. Some secure firewall set-ups refuse to accept or forward ICMP packets correctly to avert attacks. In this case, if these options were enabled on an IP tables firewall, “nothing” would actually occur when the route was blocked.

5.4 1:N Packet Aggregation

The one-to-many relationship is perhaps the most common of the fault scenarios, as it is distinctly characterised by a server or service which is used by a large number of users, suddenly terminating causing large amounts of ICMP packets to be broadcast from the machine itself or a node in the route which is within close proximity. Such examples include a a service being shut down, a server rebooting or a route failing.

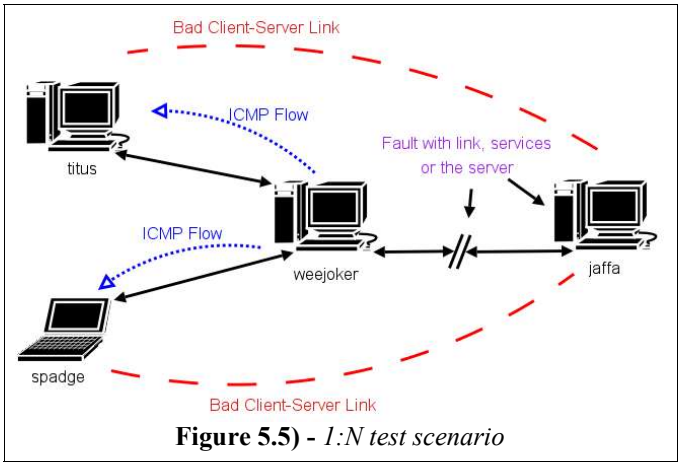
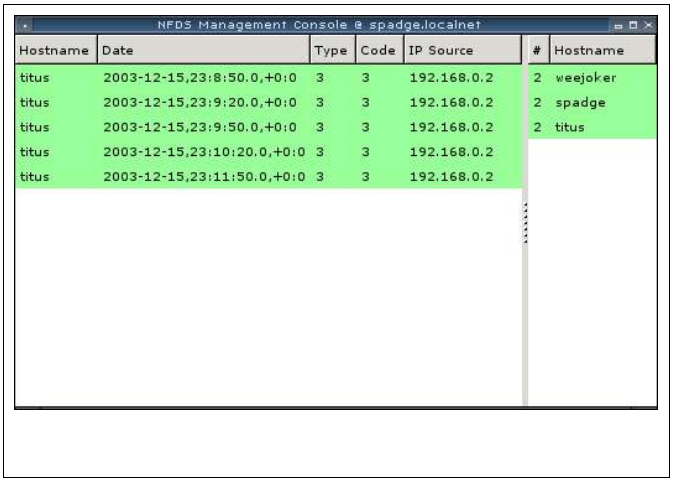


Figure 5.5) - *1:N test scenario*

To replicate this scenario, both an Apache web server and a server reboot of 'jaffa' were induced to create the fault. This would therefore cause 'weejoker' to issue ICMP packets to both 'titus' and 'spadge' when 'jaffa' was unreachable. This is a screen shot of what actually occurred:



The test for this model proved unsatisfactory, as the node 'spadge' could not receive any packets for some strange reason, but 'titus' could receive port-unreachable icmp packets when a connect was attempted to 'jaffa'. What was observed however, was that 'spadge' did occasionally receive some packets of type 11, code 0. This indicates that the connections that 'spadge' was attempting were timing out. Perhaps if this was investigated further, the cause of this “problem” may be apparent. The current guess is that 'spadge' may have cached some knowledge of the route to 'jaffa' and that may have caused the time outs. Another point of note is the rate of the packets titus was receiving: it was at a much slower rate than that of the 1:1 model. This suggests that a firewall block has a higher rate of packets emitting than a broken route, where the router does not intervene.

Further improvements for this test in the future could be:

- testing this with nodes that have exactly the same software/hardware
- testing this with a router, to see the differences of how a dedicated router detects

failures as opposed to gateway set-up used here

- try to break the routing tables by hand.

5.5 N:1 Packet Aggregation

When a service is attacked by several machines (a distributed denial of service or DDoS) or by a machine pretending to be several machines at once so to preserve is anonymity, this is classed in the many-to-one category. As it appears that several hosts are bombarding a particular host, this means that the network may be degraded itself and the majority of the packets should concentrate on one host. This time the node at the source of the "fault" is 'weejoker', as it should have degraded performance or possible service failure. The real source of the problem is usually untraceable, 'jaffa' in this instance, whom is sending out "spoofed" packets and masquerading behind these.

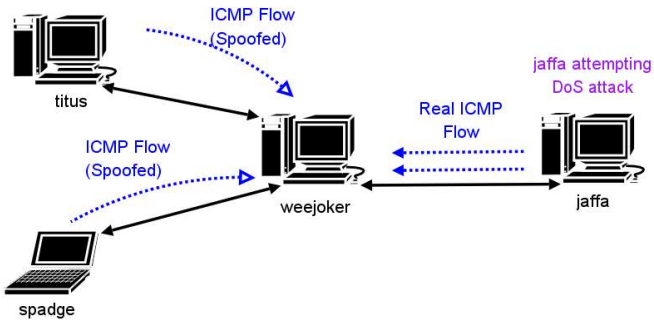


Figure 5.6) - *N:1 test scenario*

To emulate this scenario, the port-scanning utility Nmap was used to scan 'weejoker' from 'jaffa; with ping probing and spoofing the packets from a whole subnet. This should have resulted in many packets being received on 'weejoker' only. The results observed in the application successfully noted that the ICMP sniffing agent installed on 'weejoker' did indeed detect an N:1 attack, as can be seen with the following screen shot, but the result was totally unexpected:

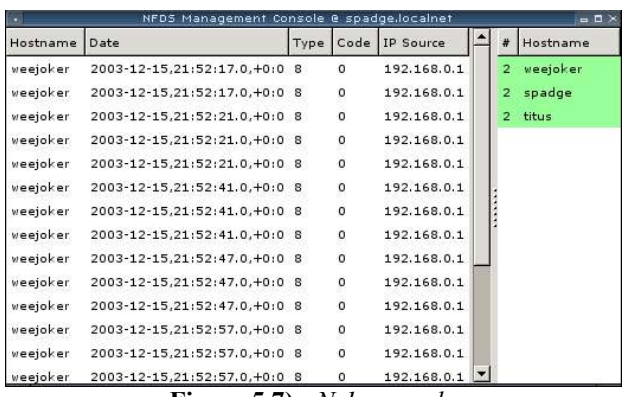


Figure 5.7) - *N:1 anomaly*

This anomaly probably occurs as a result of the packet being received on another interface rather than what the kernel routing table says, therefore the kernel devises that the only possible place the packet could originate from is itself, as it is a gateway

between the two networks and holds an IP in each (192.168.0.1 and 192.168.1.1).

5.6 Conclusions

The scenarios used to test each of the facets of the application worked well in most cases, although there was some unexpected results, perhaps due the lack of research obtained on fault-test scenarios. Here are the problems summarised:

- The N:1 model showed that if the attack appears from one interface, claiming to be from the other, the target of the attack will assume this is happening from itself.
- The 1:N model was too inconsistent on different hosts for some unknown reason. Potentially a routing cache problem on one of the hosts.

We can however draw that:

- The NIC agent works well and can record the correct status of a network interface.
- The ICMP agent can log packets correctly, providing they are not spoofed.
- The SNMP application can retrieve data successfully.
- The aggregation function works correctly, if the correct data is obtained.
- The application has the ability to portray the status of the network correctly.
- The 1:1 model works well, but only after a small amount of testing

There was however a smaller problem noting that the application slows considerably when some SNMP data cannot be collated, and this is a serious problem if the system was to deal with several nodes. In future, it would be preferable if the tests were more complex and more research done in this area. This could perhaps generate some very interesting results, therefore giving an insight as to how the system could be more finely tuned in future.

6 Conclusions

6.1 Evaluation of Achievement

During this project the aim of creating a distributed, agent-based, network fault diagnosis system was addressed. In the literature review in section 2, each of the widely available fault-finding methods were investigated to see which had the most potential in this project for recreating a network fault management system that Dupuy (1989) describes. Eventually, Kanamaru's (2000) symptom-based aggregation technique was chosen to be the most prosperous to develop in time, as it was simple and easy to implement. To deploy the fault detection technique over a large area, the SNMP [Case (1990), Stallings (1999)] standard was decided upon as the most widely available and most viable option to hand, as it was widely supported and commonplace.

Finally ICMP [Postel 1981] was observed to be the most ready source of error data, and the research discovered that this would bond well with the packet aggregation technique. To determine if a fault was due to a faulty connection or network card, the internal registers of the network card was examined to see if any possible information could be obtained regarding the link status. This was found to be possible through the MII registers [Becker] and was therefore selected as a promising alternative. The results of the research emphasises that the most suitable fault-detection methods, agent-distribution and error source was selected. This, in accordance to time constraints, meant that the chosen technologies allowed for time to understand and develop designs around these, yet allowing the opportunity to have usable test results and analysis.

In section 3, the packet aggregation technique was developed further, so that the method could be more clearly understood. This would allow the system to be built appropriately, aiming towards detecting faults in the test scenarios effectively. The use of a network cards MII registers was an attempt to aid the system into clearly defining if the fault was related to a hardware or media connection fault and the ICMP data to see if the fault was due to the loss of a service instead. This ICMP data could also determine the scope of a fault by various models, ranging from client-to-server problems, issues that affect many hosts dependant on a single central server and even network attacks by malicious actions of other computer users. The design therefore came to the conclusion that, if put into practise, the system could therefore be used to determine the scope and the possible cause of error existed on the network as a whole.

After implementing the system, the system was subject to testing through four simple scenarios. This however yielded poor results, as the tests were not complex and varied enough, and some results were inconsistent or unexpected. This is perhaps due to the lack of information available on the subject of fault modelling. The tests did however manage to expose several anomalies within the system itself however, so all was not lost. If there was an opportunity to correct those flaws, then the system could may have handled a larger, more real-life test scenario, but the software flaws were

numerous and irritating, such as:

- MIB anomalies
- Interference when NIC registers were read
- System slowdown on time outs due to lack of asynchronous behaviour

There was also further problems regarding the development of the system, where a new programming language (C) had to be learned along with the use of sockets and 3rd party libraries such as libPCap and Net-SNMP. If it was possible to redo the project at a later date, these foundation elements would have been learned prior to undertaking the project and would have resulted in more time in developing areas such as recursive packet aggregation (rather than on a single-pass) and non-blocking techniques such as asynchronous operation.

6.2 Outline of Outstanding and Future Work

Throughout the development of the fault diagnosis system, it has been impossible, due to time constraints to implement some features. Some of the areas that should have been more complete are:

- **Recursive aggregation** – The aggregation function produced was only able to analyse the biggest flaw detected with the all the ICMP packet data. The trying to apply a "divide-and-conquer" [Ohta (1997)] technique, would have resulted in a recursive algorithm. This recursive function may have taken some time to develop, as returning information from within a series of self-calling functions is quite difficult to devise without more in-depth analysis and research. Trying to differentiate between a 1:1 and a 1:N aggregation, whilst attempting to compare the severity is another research area, perhaps out with the network side and more of a human-computer interaction issue.
- **MIB flaws** – During the testing phase, it was noticed that the NIC-side of the MIB operated differently when certain parts of the ICMP pack logging was commented out (the indexes of the ICMP table). Perhaps further investigation into this shall shed light as to why this anomaly occurs.
- **Asynchronous SNMP retrieval** – Attempting to facilitate the retrieval of both the SNMP value and table data proved too complex when designed in an asynchronous manner. Passing data between the main program body and a callback function may have taken too much time to devise, so a synchronous solution was opted for instead. This synchronous solution however, is not efficient in terms of the time to collate great amounts of data, especially when some connections time-out and are non-responsive.
- **Multiple interface handling** – Adding the functionality of handling multiple interfaces may have added flexibility, but it would have taken a lot of coding to achieve this. To achieve this on each agent, it would have needed a substantial amount of research into process forking, creating and maintaining lists of network interfaces, mutual exclusion and the added problem of how each interface could be uniquely identified within the MIB.

- **Scaling of aggregation scores** – Some of the important packet aggregation models are less visible when contrasted with the less critical models. Such an example is the 1:1 model, where the only evidence is present upon two nodes. As the number of nodes in the network increases, the ability to observe this fault gets harder, as the 1:N and N:1 algorithms are visible on much more hosts. Therefore, as the network scales, it should be possible to scale up some of the models which are typically observed on a smaller scale.
- **Better fault modelling –** The lack of fault models meant that the tests used were not benchmarks, but test "tests". This meant that lots of the results returned were unexpected. Perhaps further research could be carried out in this area to determine what exactly causes a "host unreachable" response from ICMP packets and how this can be emulated.
- **Larger, more complex test scenarios** – The scenarios used within the tests are of very small scale. Due to time constraints, it was bordering near impossible to acquire machines for use within the tests. It would also have taken a considerable amount of time to set many of these up and to ensure that each node performed the same.
- **Investigation of agent performance** – Whilst testing the agents, the network media detection agent showed signs of interrupting network traffic when the network interface was polled. Therefore all agents should have been subject to a detailed analysis so that any possible performance-reducing bugs could be eliminated. This would cover things such as CPU and memory usage, hardware/software interference and the performance of the agents themselves.

6.3 Potential Enhancements

There is a great deal of potential in this project for further incorporation of new technologies and more advanced techniques. Although the research into some areas is weak at the time of writing, there are some key areas that could be added.

- **QoS Measurement** – Ensuring the Quality of Service can be maintained on a commercial network is of vital importance in today's environment. Being able to trend and predict when at network or services are reaching capacity would be of use to management and system administrators alike, so that the network bandwidth and resources could be scaled or redesigned to accommodate more nodes [Barford (2001, 2002), Trzec (2002)].
- **Knowledge-based ICMP Diagnostics** – Rather than letting the administrator try to decrypt what the exact cause of the fault may be, a system could be implemented to interpret this data automatically. A knowledge-base [Houck (1995)] set-up could be used to profile common faults and depending on the types and code of the ICMP packets that can be seen, a more exact diagnosis can take place by comparing the values seen with previous examples
- **IPv6 Integration** – Although IPv6 is not in common use at the time of writing, as time progresses it shall eventually surpass IPv4 as the standard network protocol. As IPv6 is a newly redesigned protocol, ICMP has also been redesigned too. ICMPv6 [Stevens (1998)] has a smaller, more concise set of ICMP values, therefore indicating that IPv6 may see a new trend of errors, as the IPv6 protocol inherently solves these itself.

- **Coding-based Schemes** – As stated in the literature review, coding-based schemes used for fault detection appears to be a promising field [Kliger (1995)] . With more development time, this may have been a very effective and efficient approach to diagnosing faults upon the network, especially when used in conjunction with symptom-based aggregation.
- **Wireless Computing –** Unlike IPv6, wireless computing has quickly appeared on the computing scene and its demand is steadily increasing. Due to this rapid emergence, the wireless technology is relatively unstandardised and therefore lacks fault tolerance. Therefore this would be a good area to cover with a fault diagnosis system, as it could pinpoint wireless problems with range and access point deficiencies.

7 REFERENCES

Barford P, Plonka D. "Characteristics of network traffic flow anomalies."
Proceedings of ACM SIGCOMM Internet Measurement Workshop '01. 2001; 69-73.

Barford P, Kline J, Plonka D, Ron A. "A signal analysis of network traffic anomalies."
Proceedings of ACM SIGCOMM Internet Measurement Workshop '02. 2002; 71-82.

Case J, et al. "A Simple Network Management Protocol (SNMP)"
RFC1157. 1990.

Chao CS, Yang DL, Liu AC. "A LAN fault diagnosis system."
Computer Communications. 2001; **Vol 24**, 1439-1451.

Chessa S, Santi P. "Crash faults identification in wireless sensor networks."
Computer Communications (In press). 2002; 1-10.

Deng RH, Lazar AA, Wang W. "A probabilistic approach to fault diagnosis in linear lightwave networks."
IEEE J. on Select. Areas in Communication. 1993; **Vol 11: #9,** 1438-1448.

Dupuy A, et al. "Network fault management – a user's view"
Proc. of the 1st Int. Symposium on Integrated Net. Mgmt. 1989.

Frontini M, Griffin J, Towers S. "A knowledge-based system for fault localization in wide area networks"
Integrated Network Management II. 1991; 519-530.

Fuller W. "Network management using expert diagnostics."
Int. J. Network Mgmt. 1999; **Vol 9**, 199-208.

Gay WW. "Linux Socket Programming By Example"
Que. 2000; ISBN: 0-7897-2241-0

Hong P, Sen P. "Incorporating non-deterministic reasoning in managing heterogeneous network faults."
Integrated Network Management II. 1991; 481-492.

Houck K, Calo S, Finkel, A. "Towards a practical alarm correlation system"
Proc. of the 4nd Int. Symposium on Integrated Net. Mgmt. 1995; IFIP, 226-237.

Inder R. "Experience of constructing a fault localisation expert system using an AI toolkit."
AI Applications Institute, University of Edinburgh, 1988.

Jakobson G, Weissman MD. "Alarm correlation"
IEEE Networks. 1993; **11**: 52-59.

Joseph C, Sherzer A, Muralidhar K. "Knowledge based fault management for OSI networks."
Industrial Technology Institute, Ann Arbor, 1990.

Kanamaru A, Ohta K, Kato N, Mansfield G, Nemoto Y. "A simple packet aggregation technique for fault detection."
Int. J. Network Mgmt. 2000; **Vol 10:** 215-228.

Katzela I, Schwartz M. "Schemes for fault identification in communication networks." *IEEE Transactions on Networking.* 1995; **Vol 3: #6,** 753-764.

Kernighan BW, Ritchie DM. "The C Programming Language (2nd Edition)"
Prentice Hall PTR. 1988; ISBN: 0-13-110362-8

Kliger S, et al. "A coding approach to event correlation"
Proc. of the 4nd Int. Symposium on Integrated Net. Mgmt. 1995; IFIP, 266-277.

LaBarre L. "Management by Exception: OSI event generation, reporting and logging"
Proc. of the 2nd Int. Symposium on Integrated Net. Mgmt. 1991.

Lewis L. "A case-based reasoning approach to management of faults in communication networks."
IEEE Proc. of conf. of AI applications. 1993; 114-120.

Lo CC, Chen SH, Lin BY. "Coding-based schemes for fault identification in communication networks."
Int. J. Network Mgmt. 2000; **Vol 10:** 157-164.

Mansfield G, et al. "A SNMP-based expert network management system"
IEICE Trans. Communications. 1992; **E75-B: #8**, 701-708.

Mori T, et al. "The dynamic symptom isolation algorithm for network fault management and its evaluation"
IEICE Trans. Communications. 1998; **E81-B: #12**, 2471-2480.

Muller NJ. "Improving network communications with intelligent agents."
Int. J. Network Mgmt. 1997; **Vol 7:** 116-126.

Ohta K, et al. "Divide and conquer technique for network fault management"
Proc. of the 5th Int. Symposium on Integrated Net. Mgmt. 1997; IFIP, 675-678.

Pearl J. "Probablistic reasoning in intelligent systems: Networks of plausible inference"
Morgan Kaufman. 1997.

Perkins DT. "RMON Remote monitoring of SNMP-managed LAN's"
Prentice Hall. 1999.

Postel JB. "Internet Control Message Protocol"
RFC792. 1981; 21 pages.

Reggia JP, Nau DS, Yang PY. "Diagnostic expert systems based on a set covering model"
Int. J. of Man-machine studies. 1983; **19:** 437-460.

Rouvellou I, Hart GW. "Automatic alarm correlation for fault identification"
IEEE Proc. of INFOCOM'95. 1995; 553-561.

Stallings W. "SNMP, SNMPv2, SNMPv3 and RMON 1 and 2."
Addison-Wesley. 1999.

Stevens W. R.. "UNIX Network Programming – Network APIs: Sockets and XTI (2nd Edition)"
Prentice Hall PTR. 1998; ISBN: 0-13-490012-X

Trzec K, Huljenic D. "Intelligent agents for QoS management."
AAMAS '02. 2002; 1405-1412.

Waldbusser S. "Remote network monitoring management information base"
RFC1757: 1995.

Wang C, Schwartz M. “Fault detection with multiple observers.”
IEEE/ACM Transactions on Networking. 1993; **Vol 1: #1**, 48-55.

Wang C, Schwartz M. “Identification of faulty links in dynamically-routed networks”
IEEE J. on Select. Areas in Comms. 1993; **11:** 1449-1460.

Ward A, Glynn P, Richardson K. “Internet service performance failure detection.”
Performance Evaluation Review. 1998; 38-43.

Yemini SA, *et al*. “High speed and robust event correlation.”
IEEE Communications Magazine. 1996; **Vol 34: #5**, 82-90.

Mattis P, Kimball S, MacDonald J. GTK Toolkit
Available from http://www.gtk.org

Various authors. Net-SNMP.
Available from http://net-snmp.sourceforge.net/

Jacobson V, Leres C, McCanne S. LibPCap.
Available from http://www.tcpdump.org/

Carstens T. "Programming with Pcap"
Available from http://www.iac.rm.cnr.it/~massimo/pcap.htm

Hargrave V. "LibPCap Programming Tutorial"
Available from http://vhargrave.home.attbi.com/libpcap/libpcap.html

Casado M. "Packet Capture With libpcap and other Low Level Network Tricks"
Available from http://www.cet.nau.edu/~mc8/Socket/Tutorials/section1.html

Becker D. "Understanding MII Transceiver Status Info"
http://www.scyld.com/diag/mii-status.html

8 Appendix A – Installation and Configuration of SNMP and Agents

8.1 Introduction

This is a rough installation guide for using Net-SNMP along with the userdefined agents. There is no real specification requirements, apart from installation platform being GNU/Linux based running the i386 architecture (other architectures may work) and having 4Mb free. Those who do not wish to compile with a C-compiler such as GCC, etc, can use binary packages of Net-SNMP (such as RPM's or DEB's), but it cannot be guaranteed that they are compiled correctly for use with dynamic modules.

8.2 Compilation and installation of Net-SNMP

To begin with the Net-SNMP package must be downloaded from http://net-snmp.sourceforge.net/ and should take the form of a gzipped tar-ball with the naming convention of net-snmp-X.Y.Z.tar.gz. The project used version 5.0.9 for development.

Once the tar-ball has been downloaded, it can be extracted using the line:

```
$ tar -xfvz net-snmp-X.Y.Z.tar.gz
```

When this has been done, the package is ready for configuration and compilation. It is advised that the configuration option "--enable-shared" is used.

```
$ cd net-snmp-X.Y.Z
$ ./configure -enable-shared
$ make
$ su
Enter password:
# make install
```

Should the dynamic loading of modules fail in future steps, a patch called config-dlmod is available for GNU/Linux configurations that fail to work.

8.3 Compilation and Installation of Agents

To house the agents, a directory must be created:

```
# mkdir /usr/lib/snmp/dlmod
```

After this, moving to the appropriate source directory, the agents can be compiled:

```
# cd <path-to-cd>/snmp/agents/nfdsSenseLink
# make && make install
# cd ../nfdsSniffIcmp
# make
```

8.4 Configuration

Before the system can be used there are some configuration files that need to be moved into place and the SNMP daemon restarted:

```
# cd <path-to-cd>/snmp/config
# cp snmpd.conf /etc/snmp/
# cp snmp.conf /usr/share/snmp/
# cp NETWORK-FAULT-DIAG-SYS.txt /usr/share/snmp/mibs/
# /etc/init.d/snmpd restart
```

Now the system should be ready to use.

8.5 Switching on the ICMP packet agent

To enable the ICMP packet logging agent, it must be started by hand for every use, providing it has been compiled correctly:

```
# ./nfdsSniffIcmp
```

9 Appendix B – Configuration Files

9.1 Introduction

The files and code that follow are used in the configuration and compilation of the snmp application, MIB and agents.

10 Appendix C – Source Code

10.1 Introduction

This is the source code for the agents and the application. This should be compiled directly from the Makefiles provided. Portions of Net-SNMP code may be copyright of the Net-SNMP team and may appear under the GNU General Public License.