MASTER'S THESIS

Automated Network Fault Management

by P. Viswanathan
Advisor:

CSHCN M.S. 96-3
(ISR M.S. 96-14)

The Center for Satellite and Hybrid Communication Networks is a NASA-sponsored Commercial Space Center also supported by the Department of Defense (DOD), industry, the State of Maryland, the University of Maryland and the Institute for Systems Research. This document is a technical report in the CSHCN series originating at the University of Maryland.

Web site http://www.isr.umd.edu/CSHCN/

Abstract

Title of Thesis: Automated Network
Fault Management

Name of degree candidate: Prem Viswanathan

Degree and year: Master of Science, 1996

Thesis directed by: Professor John S. Baras
Department of Electrical Engineering

With the recent growth of telecommunication networks, network fault management has gained much importance. Since it is a difficult task for human operators to manage large networks, the idea of automating some of these functions has attracted some attention. Some of the ideas proposed for automating such functions include the use of artificial intelligence techniques. Neural networks are very useful for performing analysis of large volumes of numerical data. Expert systems are useful for analyzing observed symptoms and identifying the cause by using a rule-based approach. However, research in artificial intelligence has shown that when either of these two methods is used alone, several weaknesses are observed in the resulting system. Thus, some other methodology would be required for tackling such large problems.

In this thesis, an approach involving the use of a hybrid system involving both neural networks and expert systems for performing automated network fault management is investigated.

Automated Network Fault Management

by

Prem Viswanathan

Thesis submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1996

Advisory Committee:

Professor John S. Baras, Chairman/Advisor
Professor Michael Ball
Professor Mark Shayman

Acknowledgments

I wish to express my thanks to some of the many individuals who made this work possible. Foremost among these is my advisor, Professor John Baras, who gave me the opportunity to work on a very interesting and practical problem as part of the research for my thesis.

I am also grateful to Professors Michael Ball and Mark Shayman for kindly consenting to join the defense committee and review this thesis.

This work was supported by Loral Western Development Laboratories and the Institute for Systems Research through a graduate research assistantship.

Finally, I wish to thank my family and friends for their support and encouragement.

Table of Contents

List of Tables

List of Figures

Automated Network
Fault Management

Prem Viswanathan

August 26, 1996

This comment page is not part of the dissertation.

Typeset by LaTeX using the `dissertation` class by Pablo A. Straub, University of Maryland.

Chapter 1

Introduction

During the last decade, there has been a tremendous growth in the services offered to users of communication networks. The increasing demand on the part of such users has led to the evolution of large networks. In order to maintain such large networks and to provide services to users in an efficient manner, good network management techniques are necessary. In the ISO model for network management, there are five functional areas, listed below:

1. Accounting Management.
2. Configuration Management.
3. Fault Management.
4. Performance Management.
5. Security Management.

Our focus is in the area of fault management. Fault management is very important because network service providers might lose customers if they do not

provide efficient service. For example, if users experience long delays in accessing information from a site, it could be very frustrating. Some of the functions of fault management include detecting, isolating, and repairing problems in the network. It also deals with the ability to trace faults, given many alarms in the system. Furthermore, it is also concerned with the use of error logs, and tracing errors through the log reports. One of the problems faced by network control centers (which manage such networks) is that of handling extremely large volumes of data dealing with the performance of the networks. It is often very difficult to determine the existence and location of a problem in the network if a network operator (at the network control center) would have to analyze the data. The volume of such data would make the task of finding the problem a very time-consuming process. However, unlike some of the other network management functions listed in the ISO model, in fault management, speed is very crucial and recovery from a problem has to occur quickly. If the downtime for systems is high, it could lead to a substantial financial loss. In order to be able to handle huge amounts of data quickly, we used some ideas from artificial intelligence, with neural networks and expert systems. Such ideas have been applied with some success in chemical engineering, in the management of chemical processes. We have applied some of those ideas to the network fault management scenario. By using artificial intelligence techniques to automate some of the fault management functions, network operators can provide assistance in other areas.

In this project, a prototype system has been built to perform *automated network fault management*. Our system consists of a X.25 packet network simulated using OPNET, a communication network simulation tool from MIL3, and a neural network/expert system module developed using the G2/Neuronline package

from Gensym Corporation. We performed SNMP (Simple Network Management Protocol) monitoring of variables in order to aid in the fault management process. Our approach consists of a **proactive** component and a **reactive** component in the fault management process. The former is achieved using a minimum cost routing algorithm. The latter is achieved by the use of a hybrid neural network/expert system architecture. These points will be explained in greater detail in later portions of this thesis.

The rest of this thesis is structured as follows. In Chapter 2, a detailed description of the X.25 network simulation is provided. The assumptions used, together with the features present in the simulation for simulating the occurrence of faults, are all described. In Chapter 3, the SNMP variables being logged in the X.25 simulation, together with the SNMP traps, are described. In Chapter 4, the neural network and expert system components are described. Training and testing results are presented. The kind of queries used by the expert system are also described in detail. In Chapter 5, the results from Chapter 4 are summarized in a step by step, concise format. Finally, in Chapter 6, some conclusive remarks are presented.

Chapter 2

Description of Network Topology in OPNET

The network that we have designed in OPNET is based on the X.25 protocol. Since this is a prototype system, our network has the option of having up to 10 users. However, more users can be easily added. Each user corresponds to a Data Terminal Equipment (DTE), connected to a Data Communications Equipment (DCE). A DTE usually corresponds to a dumb terminal, while a DCE usually corresponds to a modem. Thus, having 10 users implies having 10 DTE/DCE pairs, where each DTE can have several logical channels. Each DTE can handle 2 applications, thus making it possible to run up to 20 applications at a time. There are both permanent virtual circuits (PVCs) and virtual calls. Two PVCs have been predefined. In addition to the DTEs and DCEs pertaining to the X.25 model, there is also a SNMP manager, details of which will be provided in Chapter 3. A diagram illustrating the network topology is presented in Figure 2.1.

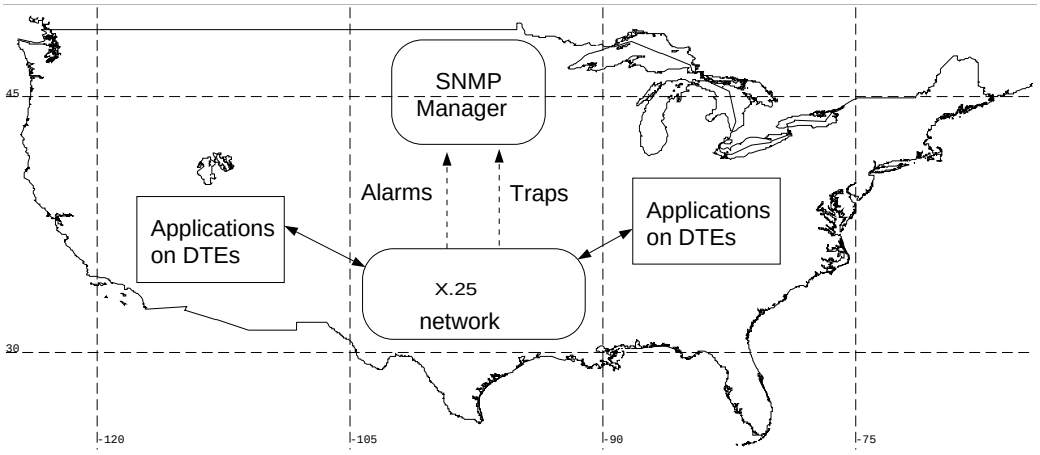

Figure 2.1: Diagram illustrating the network topology.

2.1 The X.25 Cloud

In the X.25 scenario, information travels from a DTE to a DCE, which in turn sends the information onto the X.25 network. This is often referred to as the X.25 cloud. In our simulation, the X.25 cloud consists of 15 nodes (or switches — these two terms shall be used interchangeably in this document) used to transmit the packets in a store-and-forward manner. These 15 nodes are grouped into 3 subnetworks, where each subnetwork consists of 5 nodes. A typical subnetwork is shown in Figure 2.2.

The five nodes in each subnetwork are not fully interconnected, i.e. every node does not necessarily have a direct link to every other node. However, there are several paths between one node and another, thus providing redundancy in the design to recover from failures at certain nodes. The main difference between subnetworks is the difference in speed for the links connecting the nodes. Thus, different subnetworks have links with different speeds. A DTE is connected to a DCE which is then connected to one of the nodes in the cloud. Each node

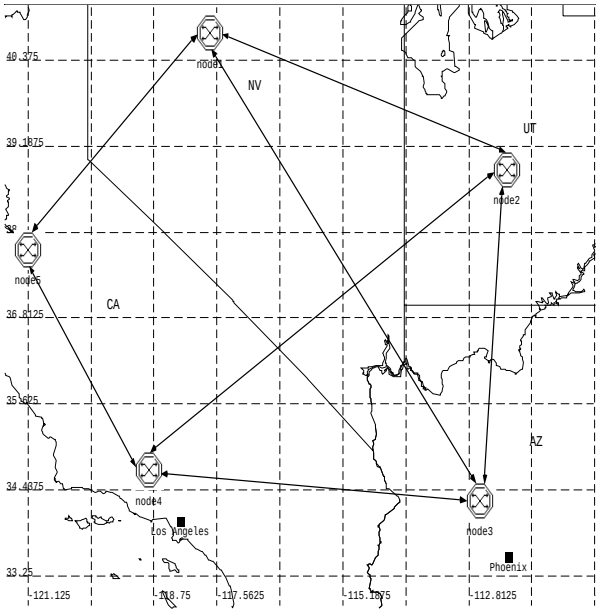

Figure 2.2: Diagram illustrating a typical subnetwork.

consists of point-to-point transmitters, point-to-point receivers, one queue for all the input links, and one queue for each output link. The internal structure of a typical node is shown in Figure 2.3.

2.2 Assumptions about the model

1. Each application generates packets using a Markov Modulated Poisson Process (MMPP). However, the calls are exponentially distributed. Therefore, *within each call*, the packets are sent in a random fashion, as modeled by the MMPP source. The source sends packets whose sizes are fixed. The MMPP source is used in order to simulate a bursty traffic model for data. Details about the MMPP model are provided in the next section.

2. Amount of data transfer. This is established by a random number generator that determines the number of paragraphs of text to be transferred

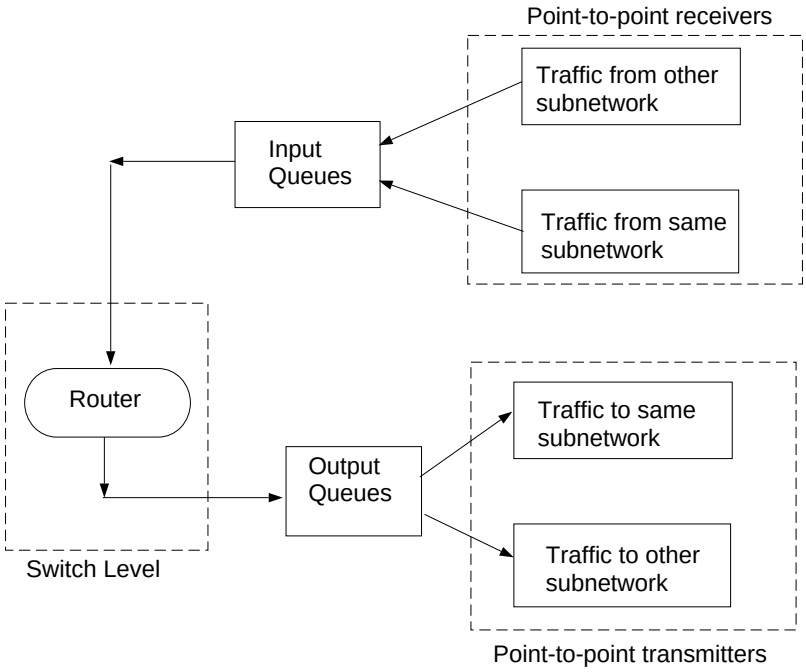

Figure 2.3: Diagram illustrating the internals of a typical node in the subnetwork.

from the source DTE to the destination DTE.

3. Each packet has a priority of 0 or 1, depending on the user generating the packet. A packet with priority 0 has lower priority than a packet with priority 1. This is used to allow for a real scenario in which a network provider might have several kinds of customers.

4. The input and output queues (shown in the figure) have finite capacity and fixed service rate that are user-specified.

5. Each DTE has a destination assigned to it. Thus, there are a collection of source/destination pairs. Each source DTE is connected to a certain subnetwork and the destination DTE could either be in the same subnetwork or in a different subnetwork.

6. Queueing with Priorities. When packets arrive at a node through the input

links, they are inserted into an input queue, waiting to be served. All the packets (arriving from the various nodes via the input links) are inserted into one queue. The queueing mechanism that is used is a preemptive scheme. Therefore, if a queue is non-empty, and a packet of priority 1 arrives, it will get inserted behind all the other priority 1 packets currently in the queue, but ahead of all the packets of priority 0. Furthermore, if there are no packets with priority 1 waiting in the queue, and a packet of priority 1 arrives while a packet of priority 0 is currently being served, its service will not be interrupted by the priority 1 packet that has just arrived.

2.3 The Markov Modulated Poisson Process (MMPP) Source

In this network, we are generating packets according to an MMPP model. In this model, we are using a 3-state Markov chain, where each state in the Markov chain represents a Poisson process with a certain rate. This is shown in Figure 2.4. This routine is written in C and is incorporated into the OPNET source code. In the program developed in OPNET, the transition probabilities p_{ij} between the different states in the Markov chain are specified by the user, together with the rates, λ_i, in each of the states in the chain.

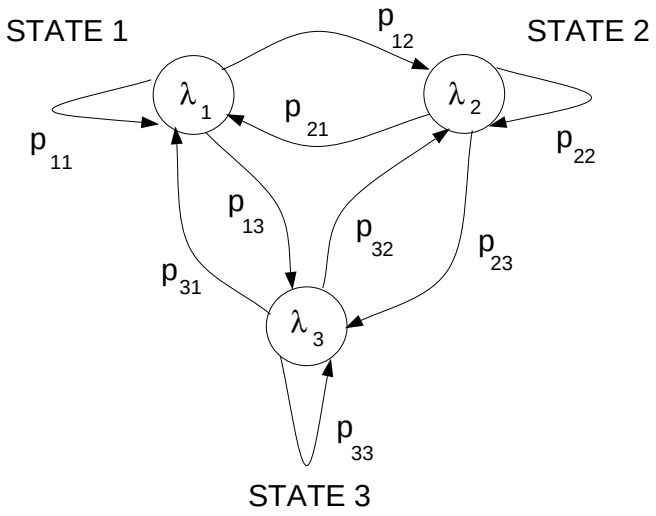

Figure 2.4: Diagram illustrating the 3-state Markov chain used to simulate the Markov Modulated Poisson Process (MMPP) traffic source.

2.4 A Minimum Cost Routing Algorithm

When a packet is created by an application running on a DTE, a source to destination pair is assigned to it. Based on this pair, a route is selected from the routing table and is assigned to the packet based on a minimum cost routing algorithm. An example of how this algorithm works is shown in Figure 2.5.

In this figure, the numbers next to each link represent the cost of that link. A packet going from Node A to Node X will follow the path A-C-D-X and not A-B-X because the former has a cost of 1.5 + 2.5 + 3.0 = 7.0 while the latter has a cost of 6.0 + 5.5 = 11.5. At the start of the simulation, all the links have a cost of 0.0 assigned to them. As the simulation progresses, this cost is updated periodically by relating the cost of the link to the utilization on the link using the following cost function:

$$c_i = (1 - \alpha)c_{i-1} + \alpha \frac{1}{1 - \rho_i}$$

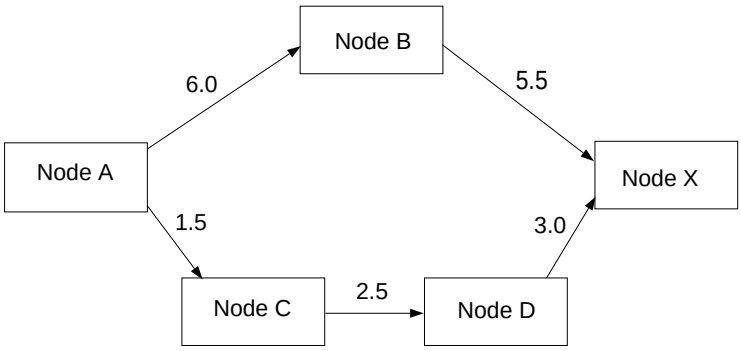

Figure 2.5: Diagram illustrating the minimum cost routing algorithm.

where c_i is the cost of the link at the i^{th} time instant (when the data is sampled), c_{i-1} is the cost of the link at the previous time instant, α is a weighting factor between 0 and 1 (inclusive), and ρ_i is the utilization on the link at the i^{th} time instant. The weighting factor, α, is used in order to take into account the dynamics of the network. By taking a portion of the previous cost and a part of the current utilization, the new cost is computed. The choice of α is left for the network designer. In our simulations, we chose α to be a number greater than 0.5, thus assigning more weight to the current value of the utilization (and therefore less importance to the past).

When the utilization of a link increases, the cost of the link increases also (though not in a linear fashion). As a result, the traffic will be re-routed through links that are relatively underutilized. Therefore, there is a notion of "self-healing" that takes place within the X.25 cloud itself. This is the *proactive* component of our design.

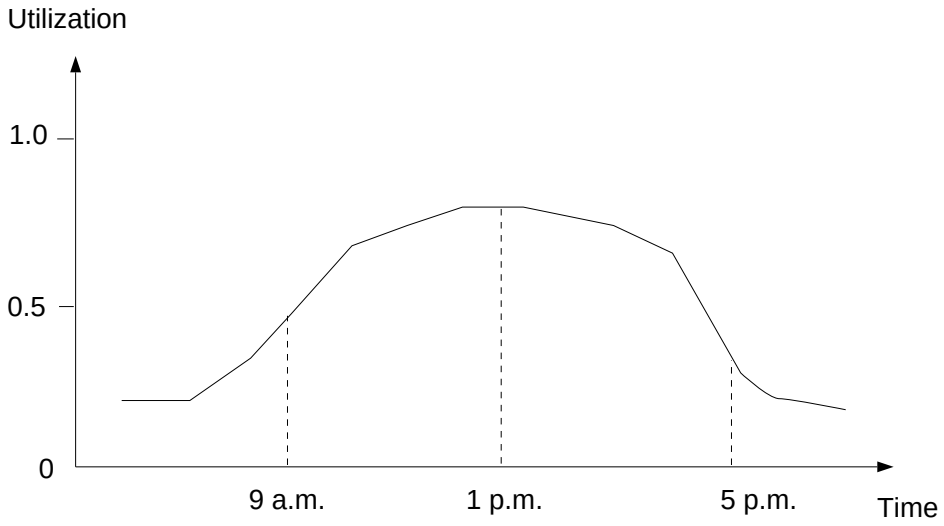

Figure 2.6: Diagram illustrating how traffic would fluctuate during a typical business day.

In addition to depending on the source and destination addresses, this routing algorithm also depends on the maximum number of hops allowed. This parameter is specified by the user. Thus, if the maximum number of hops is 2, then path A-C-D-X would not be considered in Figure 2.5.

2.5 Simulation of Varying Traffic Patterns

When running a simulation for T seconds, we are varying the traffic in such a way that there are periods when traffic is high and other periods when there is less traffic. A possible example of this is illustrated in Figure 2.6.

For example, if T = 1200 seconds, we start simulating with a small number of applications and at t = 400 s, we bring the network to maximum traffic. At around t = 750 s, we remove a couple of applications, reflecting the possibility

that the real users might be away from work, e.g. at lunch. Then, by reintroducing other applications, we reach peak traffic again. At around t = 950 s, we reduce the traffic again, and complete the simulation. By varying the traffic in such a manner, we believe this would be a fairly good model for the network traffic in a 24-hour period. Note that during the simulation, we never bring the traffic completely to zero since there will usually be some traffic on the network at all times. Examples of such traffic include backups by system administrators, execution of scripts specific to the operating system (such as UNIX, DOS, etc.), automatic logging of network performance data, and so on. Depending on the computing resources available to run the simulation, one could use different lengths of time to correspond to a 24-hour period. Our simulations were conducted on Sun Sparc stations and thus, we were able to run them for 24 hours of simulation time (which took anywhere between 1 and 2 hours, depending on the load on the network at the time the simulation was running).

2.6 Data Generated by the Network

The performance data being collected consists of statistics about the following parameters:

1. Blocking of packets
2. Queue sizes
3. Packet throughput from all the applications
4. Utilization on links connecting subnetworks
5. End-to-end delays experienced by packets

The data is sampled periodically for the first 4 parameters over a certain time window. This period can be specified by the user, based on how often such measurements need to be made. In our simulations, this interval is 50 seconds of simulation time. The delays are computed every time a packet reaches its destination. When a packet enters the X.25 cloud (from the DCE) at a certain node, as shown in Figure 2.2, it is time stamped. Upon reaching the destination node (before being passed to the DCE at that node), the difference is determined between the current simulation time and the instant when the packet was time stamped. This difference corresponds to the end-to-end delay experienced by the packet in traveling through the X.25 network. Thus, end-to-end delays are computed *within the X.25 cloud* without including the DTE-DCE time at the source and the DCE-DTE time at the destination.

In addition to the above, we also have SNMP variables being monitored and traps have also been implemented in our simulation. These points are discussed in greater detail in Chapter 3.

2.7 Fault Scenarios

The modeling of faults is done as follows. We define a "normal" state in the network, where "normal" refers to levels of traffic flow that are not unusually low or high, e.g link utilization between 0.20 and 0.70. Then, a set of fault scenarios are modeled and used to train the neural network system. By training the neural network to understand a normal state of operation, it would then be able to recognize abnormal states also. The fault scenarios that we have simulated are the following:

1. Reducing switch capacity, i.e. dropping the service rate. This would affect blocking of calls and response times for applications.

2. Increase the (normal) packet generation rate of a certain application (e.g. 3 times the original amount of traffic).

3. Disabling of certain switches in the X.25 cloud. This would cause re-routing of calls around other (working) switches.

4. Disabling certain links.

These scenarios are used for training the G2/Neuronline model. The OPNET simulator provides the capability of altering the attributes of network elements during the simulation. Examples of such attributes include link speeds, service rates of buffers, traffic generation rates, and so on. Using this feature, we are able to model scenarios that could represent faults within the network.

2.8 Alarms

A method for simulating the occurrence of alarms is also incorporated in the simulation. The alarm contains information regarding the severity of the problem, the location of the event (i.e. which node in which subnetwork), and a threshold. The severity is defined as shown in Table 2.1.

The decision of whether or not to send an alarm is determined by examining the sampled data over a user-specified time window. If, for example, the queue size for a particular queue is above a certain minimum threshold, then a counter is incremented. Each time this parameter is sampled *during that time window*, the counter will be incremented if it exceeds that threshold. If the value of the

Alarm Information **Severity Codes**	
Alarm Code	Severity Level
critical	5
major	4
minor	3
warning	2
informational	1
cleared	0
Alarm codes associated with performance data	
Link utilization	0, 1, 2, 3, 4, 5
Blocking	5
Queue size	4, 5
User throughput	4, 5

Table 2.1: Alarm Information.

counter exceeds a certain number, then that value is incorporated in the alarm packet as the threshold. These counters are used to allow for the possibility of severe fluctuations in the sampled data. Thus, if one sample of a certain parameter has a high value, but a few samples before it and after it appear normal, then an alarm would not be sent. This avoids the generation of unnecessary alarms, causing confusion at the network control center.

Alarms have been defined for all the performance measures, except for end to end delay. For link utilization, all the levels of severity (of the alarm) are used. For queue size, we are only using levels 4 and 5. The same is true for user throughput. For blocking of packets, we are generating an alarm (of severity level 5) every time blocking is observed (for a certain number of times over the time window). This information is summarized in Table 2.1.

Chapter 3

SNMP Monitoring

In much of the literature that was reviewed, there had been little mention regarding the use of SNMP variables to perform fault management. In our approach, we are logging statistics pertaining to SNMP variables based on the RFC 1382, "SNMP MIB Extension for the X.25 Packet Layer" and we have also installed SNMP traps, as described in the following sections.

3.1 Logging of SNMP variables at the X.25 packet layer

The following list of variables was extracted from RFC 1382 and are being logged during the simulation:

1. x25StatInCalls. This is the number of incoming calls received.

2. x25StatInDataPackets. This is the number of data packets received.

3. x25StatOutCallAttempts. This is the number of outgoing calls attempted.

4. x25StatOutDataPackets. This is the number of data packets sent by the Packet Layer Entity (PLE).

5. x25StatOutCallFailures. This is the number of call attempts which failed. This includes calls that were cleared because of restrictive fast select.

6. x25StatRestartTimeouts. This is the number of times the T20 restart timer has expired. This timer specifies the length of time that a DTE will wait for a response to a *restart request* packet. The default value for this timer is 180 seconds. The restart might occur as a result of a severe problem, such as a crash in the network.

7. x25StatCallTimeouts. This is the number of times the T21 call timer has expired. This timer specifies the length of time that a DTE will wait for a response to a *call request* packet. The default value for this timer is 200 seconds.

8. x25StatResetTimeouts. This is the number of times the T22 reset timer has expired. This timer specifies the length of time that a DTE will wait for a response to a *reset request* packet. The default value for this timer is 180 seconds. Reset is used only during a data-transfer state. Reset procedures may be necessary when problem conditions arise.

This subset of variables were chosen from the RFC because they are helpful in identifying faults that could occur in the X.25 simulation. The variables are logged on a "per DTE" basis and not on a "per logical channel" basis. This is implemented by assigning IDs to each DTE. For example, if we have a DTE in California with one application (with ID 10) running on it, the ID for this DTE

would be the ten's digit, i.e. 1. If we have another DTE in Maryland with 2 applications running on it (with IDs 20 and 21), the ID for that DTE will be the ten's digit of its applications, i.e. 2. The reason for logging the data on a "per DTE" basis is so that the volume of information used by the expert system can be reduced. This would not be the case if the information were monitored on a "per logical channel" basis. In this case, the volume of information would be overwhelming and it would take the inference engine of the expert system much longer to arrive at a conclusion. This would not be good for a function such as fault management where speed is a crucial factor.

3.2 SNMP Traps

In addition to the above, we also have the facility for agents to send traps to a manager when something goes wrong. Here, an agent refers to a node in the X.25 cloud. This manager is designed to manage the switches in the X.25 cloud. It does not receive traps from the DTEs or DCEs in the network. According to RFC 1215, "A Convention for Defining Traps for use with the SNMP", there are six basic types of traps, together with a seventh *enterprise-specific* trap. These are reproduced below for convenience:

- coldStart(0). "A coldStart trap signifies that the sending protocol entity is reinitializing itself such that the agent's configuration or the protocol entity implementation may be altered."

- warmStart(1). "A warmStart trap signifies that the sending protocol entity is reinitializing itself such that neither the agent configuration nor the protocol entity implementation is altered."

- linkDown(2). "A linkDown trap signifies that the sending protocol entity recognizes a failure in one of the communication links represented in the agent's configuration."

- linkUp(3). "A linkUp trap signifies that the sending protocol entity recognizes that one of the communication links represented in the agent's configuration has come up."

- authenticationFailure(4). "An authenticationFailure trap signifies that the sending protocol entity is the addressee of a protocol message that is not properly authenticated... "

- egpNeighborLoss(5). "An egpNeighborLoss trap signifies that an EGP neighbor for whom the sending protocol entity was an EGP peer has been marked down and the peer relationship no longer obtains." This means that the exterior protocol gateway (EGP) is down.

- enterpriseSpecific(6). This can be used to define additional trap messages as needed.

In our simulation, we have implemented traps 2, 3, and 6 above. Trap 4 has not been implemented since we are not considering the modeling of gateways in the simulation. In our implementation of trap number 6 we have, among others, two fields in the packet that are for the trap code and fault code, respectively. In the case of a linkDown or linkUp trap, the trap codes are 2 and 3, respectively. For the other traps, the trap code is 6. The fault code will be a unique code for each type of fault scenario, as listed in the section "Fault Scenarios" in Chapter 2. Thus, there could be two traps with 6 as the trap code, but with 10 and 20 as the

SNMP Trap Information		
Trap Type	Trap Code	Fault Code
Reduced Switch Capacity	6	10
Switch Capacity Back to Normal	6	20
Node Disabled	6	50
Node Enabled	6	60
Link Down	2	70
Link Up	3	80

Table 3.1: Trap types, trap codes, and their respective fault codes.

respective fault codes. These fault codes are listed in Table 3.1. In addition to the trap and fault codes, each trap that is generated also includes the subnetwork ID and the node ID, indicating the sender of the trap.

Chapter 4

Expert Systems and Neural Networks

4.1 OPNET/Neuronline Interface

The data from the X.25 simulation in OPNET is gathered in a flat file and stored in an ORACLE database. The data is then read by G2 and Neuronline, where the former is the expert system and the latter is the neural network component. From the literature reviewed, we chose radial basis function networks (RBFN) as the neural network architecture for conducting classification. In implementing our system, we are using a combination of both neural networks and expert systems. A brief description of RBFN networks is provided in the next section.

4.2 Radial Basis Function Networks

In much of the research done on fault diagnosis, primarily in the area of chemical processes, the research has focused mainly on feedforward networks with sigmoidal transfer functions, commonly called backpropagation networks. One of the drawbacks of this architecture is that it generates false classifications since it cannot handle novel cases and it arbitrarily classifies areas that are not covered

by the training data.

Recently, researchers have been looking at using radial basis function networks for handling classification problems. RBFNs are three-layered networks, with an input layer, a hidden layer, and an output layer. Unlike backpropagation networks, RBFNs use Gaussian transfer functions, with one per hidden node (in the hidden layer). Thus, each radial basis function unit represents a unique local neighborhood in the input space. The hidden layer activations are determined by the Euclidean distance to the center of each Gaussian. This results in the hidden nodes having (spherical or elliptical) regions of significant activation. Such finite bounding of the activation regions is why RBFNs are able to detect novel cases. Another advantage of RBFNs is that they require less (typically an order of magnitude) time for training compared to backpropagation networks. However, they have a slower run-time execution.

The training of RBFNs is done in three stages. In the first stage, the center of each of the radial basis function units is determined using the k-means clustering algorithm. This is an unsupervised technique that places unit centers centrally among clusters of points. In the second stage, the unit widths are determined using the nearest neighbor technique, which ensures the smoothness and continuity of the fitted function. In the final stage, the weights of the second layer of connections are found using linear regression.

4.3 Using RBFN for network monitoring

One of the most crucial elements in performing fault management of networks is *speed* for fault detection, fault location, and identification of the type of fault.

For managing the X.25 network, we are using a *hybrid* architecture of neural networks and expert systems to perform the fault management functions. Specifically, we are using the radial basis function networks to analyze the *performance* data being generated by OPNET. There is one RBFN for each subnetwork. The possible outputs for the neural network are the different classes of faults that could occur in the X.25 network (as listed in Chapter 2). Thus, when a fault occurs within a certain subnetwork, the RBFN assigned to monitor that subnetwork will alert the network operator that a fault of a specific class (e.g. disabled node) has occurred. However, this will not inform the operator of the *location* of the fault. Thus, in the example above, the operator would know that a node in a specific subnetwork was disabled, but he/she would not know *which* node was disabled. Then, based on the outcome of the neural network, appropriate action is taken by the expert system. The expert system uses information about alarms and SNMP traps, together with the SNMP variables which we chose from RFC 1382, to make its conclusions regarding the possible location and cause of the fault. There will be special rules to handle disabled nodes, others to handle failed links, and so on. These rules will be described in later sections of this document.

4.4 Fault Detection and Fault Diagnosis

4.4.1 First Level of Diagnosis: Neural Networks

In our approach, we are using one radial basis function network for each subnetwork in the X.25 cloud. In Figure 4.1, the setup of the *training* phase for a specific neural network is shown. The original data is the performance data ob-

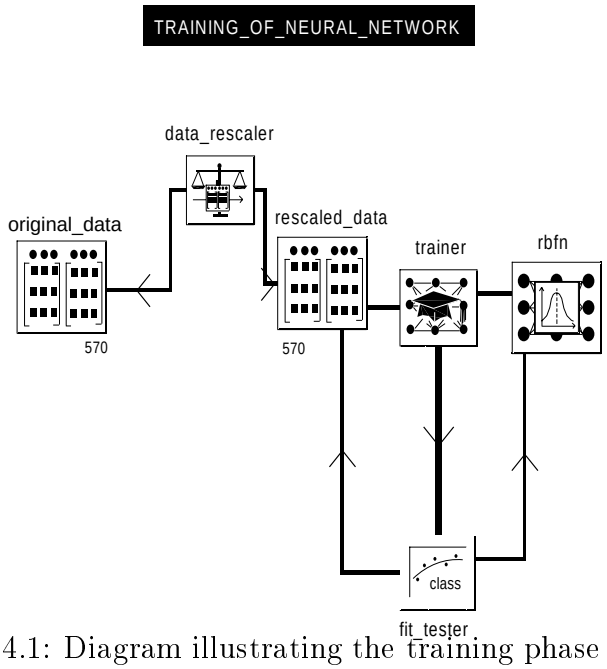

Figure 4.1: Diagram illustrating the training phase of a RBFN.

tained directly from the network. This data is then scaled using a data rescaler, which was configured to use zero mean, unit variance scaling on the input and no scaling on the output. The scaled data is then used by the trainer to train the RBFN. A fit tester is also available. The criterion chosen for the fit tester is "fraction misclassified". Thus, the output of the fit tester is a number between 0 and 1, reflecting how accurately the samples of data are classified.

The neural network has spherical nodes for its hidden layer. The number of hidden nodes per class is a parameter that does not have a unique answer. This number has to be chosen through trial and error, after several training sessions, until the desired performance is achieved. During the experiments, it was found that as the number of hidden nodes increased, the fit tester error decreased (though not linearly), thus implying that there was a better fit of the data by the neural network. However, a higher number of hidden nodes also meant a longer training period. The training of the neural networks is affected by the following factors:

1. The quality of the input data and how well it reflects the conditions of the X.25 network. If the input data is not good, the neural network will not be trained well.

2. The number of hidden nodes in the hidden layer of the RBFN. More hidden nodes usually translates to a better fit of the data, accompanied by longer training periods.

3. The number of input variables that are supplied to the neural network. In our case, we supplied the utilization levels on all the links (in both directions), the queue sizes, and the measured packet throughput at each node.

4. Is the data distinct? If the data for different fault classes is not clearly distinct, then the neural network will have a higher percentage of misclassification.

Since a neural network observes patterns and makes inferences based on those patterns, similar patterns for different fault classes would lead to misclassifications. In Figure 4.2, Figure 4.3, and Figure 4.4, the average queue sizes at the input buffers are shown for 3 nodes in a subnetwork. These graphs represent the average queue sizes under normal traffic. In Figure 4.5, Figure 4.6, and Figure 4.7, the average queue sizes at the input buffers are shown for the same 3 nodes with node 4 being disabled at t = 1400 secs. Here, we note that nodes 1 and 2 have higher buffer sizes after node 4 is disabled. This occurs as a result of re-routing of packets. In Figure 4.8, Figure 4.9, and Figure 4.10, the queue sizes at the input buffers are shown for the same 3 nodes with a user at node 1 generating excess traffic, starting at t = 1500 secs. Here, we note that all

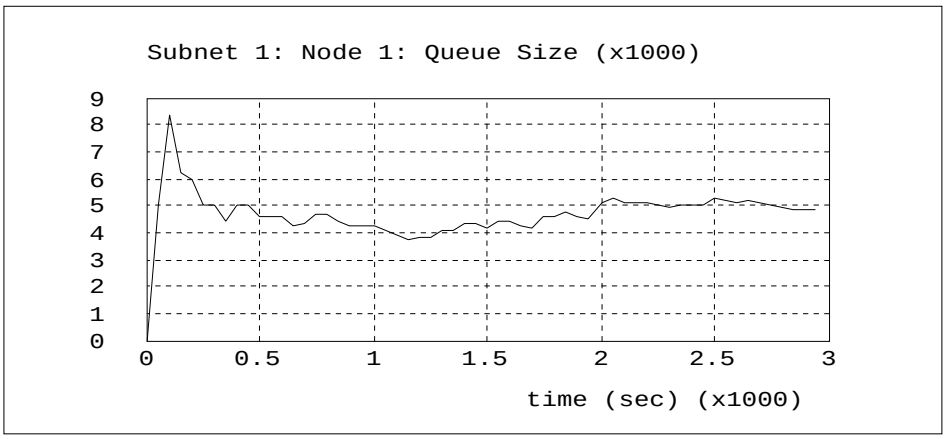

Figure 4.2: Queue size in bits for normal traffic.

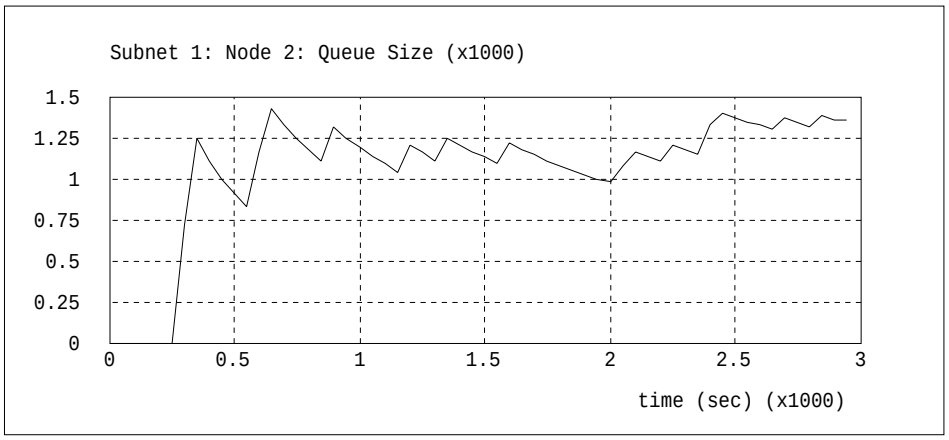

Figure 4.3: Queue size in bits for normal traffic.

three nodes have higher buffer sizes from the time the user connected to node 1 started generating the excess traffic. In the cases outlined above, there are certain distinct patterns that help the neural network to identify the different cases. However, there are other instances when it is more difficult. For example, a link failure and a node failure both lead to re-routing of traffic. If the samples of training data are small, it is very difficult for the neural network to distinguish between a node failure and a link failure, simply by analyzing the re-routing that occurs. Thus, more data is needed to distinguish between the re-routing that occurs in these two cases in order to have a small percentage of misclassification.

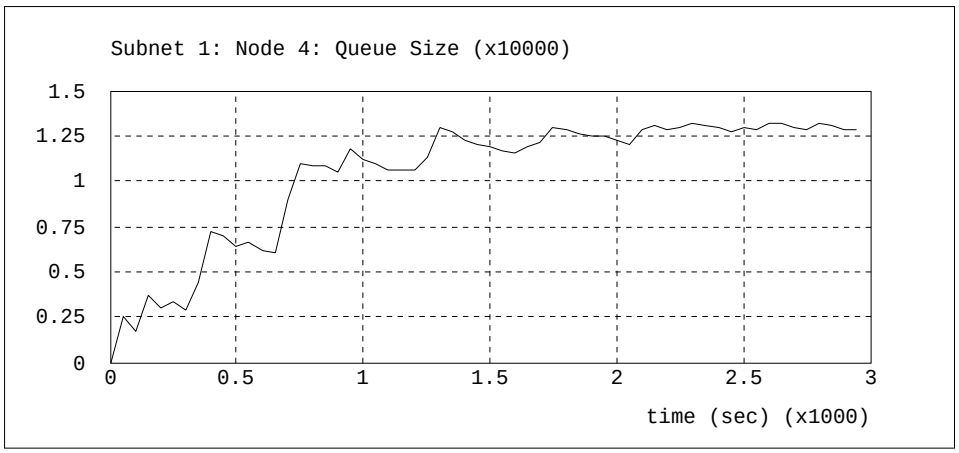

Figure 4.4: Queue size in bits for normal traffic.

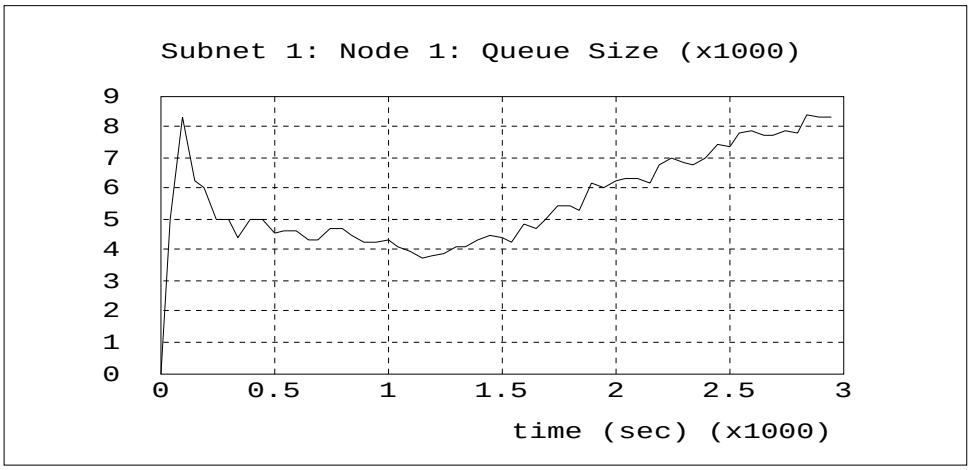

Figure 4.5: Queue size in bits with node 4 disabled.

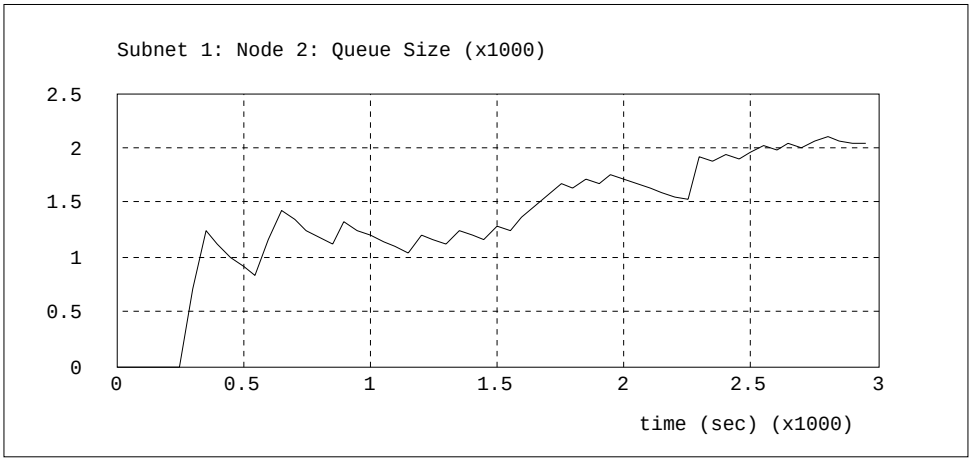

Figure 4.6: Queue size in bits with node 4 disabled.

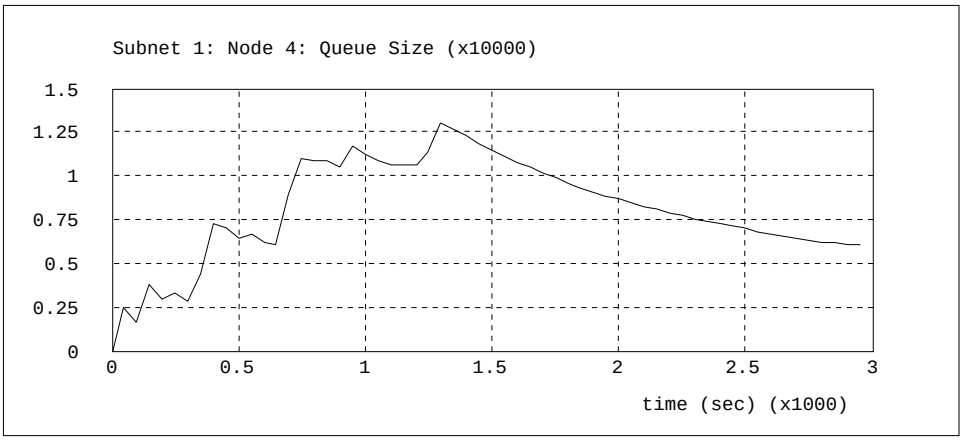

Figure 4.7: Queue size in bits with node 4 disabled.

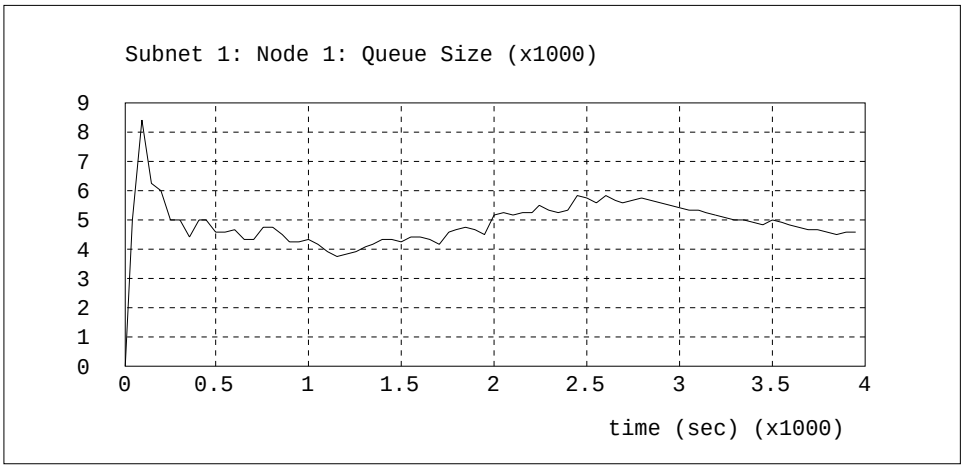

Figure 4.8: Queue size in bits with excess throughput at node 1.

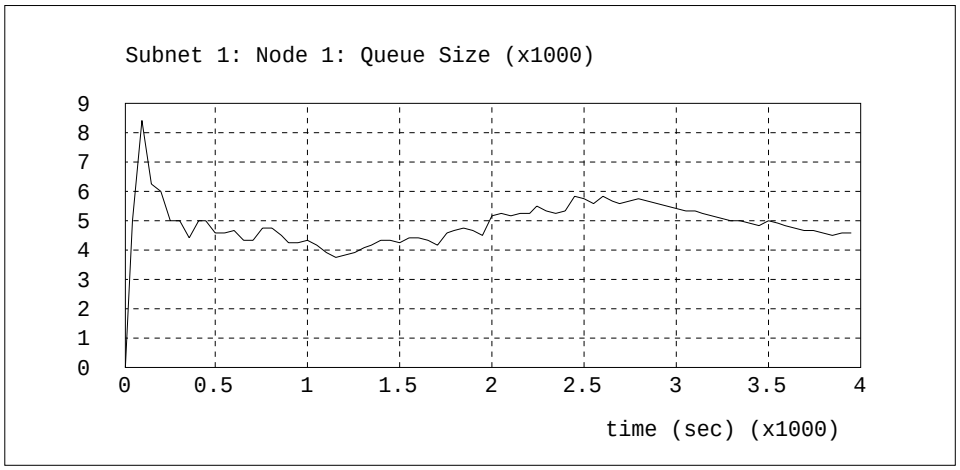

Figure 4.9: Queue size in bits with excess throughput at node 1.

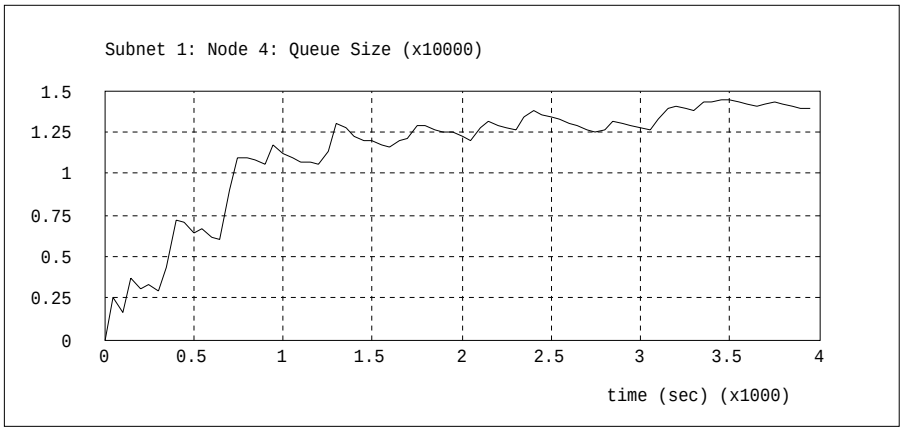

Figure 4.10: Queue size in bits with excess throughput at node 1.

In Figure 4.11, the *testing* phase of the RBFN is shown. Note that this is the testing phase for a laboratory environment only. In an actual network control center, this would be the setup used for monitoring the network. The data sampled from the network in real time is input to the block labeled "testing dataset". This data is read by a data set reader and passed to a data pair divider. The latter accepts a pair of vectors, X and Y, as input. In this case, since we are dealing with testing in a laboratory environment, X is the performance data from the network and Y is the class that it belongs to (e.g. normal state, disabled node, and so on). Since we want the neural network to be able to identify the class of fault, we reject the Y vector and retain the X vector only. This vector is scaled using the input scaler (obtained from the training phase) and then supplied to the RBFN, whose output is then observed using the output classifier. The output will be the fault class determined by the RBFN.

Several different tests were conducted to develop a better understanding of how the neural networks were trained. In the first test, we considered 3 classes (normal, disabled node, and excess user traffic). We used 180 samples of data for the "normal" class and 90 samples for each of the other two classes. The

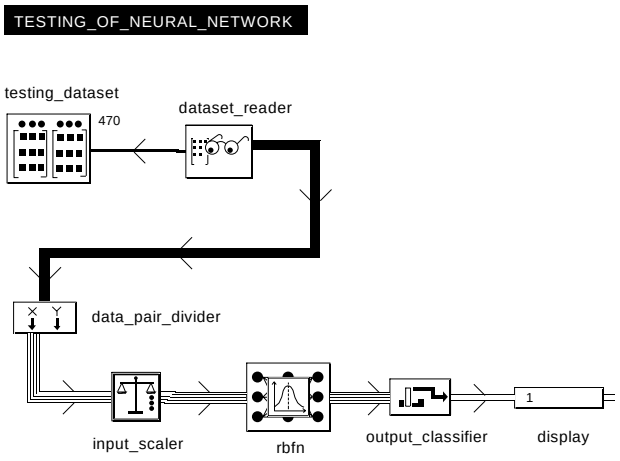

Figure 4.11: Diagram illustrating the testing phase for a RBFN.

results are shown in Table 4.1.

In the second test, we repeated the first test but changed the number of samples of training data to 180 samples per class and retrained the RBFN networks. The results for this case are shown in Table 4.2. Thus, from Table 4.1 and Table 4.2, we see that for these 3 classes, with more data points per class, the total number of hidden nodes decreases for a certain range of error values for the fit tester.

In the third test, we considered all five classes of faults and trained the RBFNs with different sample sizes. We first trained the RBFN with 150 samples for the "normal" class and 80 samples for each of the other fault classes, giving a total of 470 points in the training set. The results for this case are shown in Table 4.3. By looking at the last two columns in each row, it is observed that the percentage error is higher for those two fault classes. This provided the motivation for the next test.

Neural Network Training: 3 Classes				
Total Hidden Nodes	Fit Tester Error	Perc. Error Normal State	Perc. Error Disabled Node	Perc. Error Excess User Throughput
45	0.26	0.15	0.15	0.13
60	0.24	0.11	0.10	0.09
66	0.23	0.09	0.09	0.07
75	0.20	0.06	0.07	0.04

Table 4.1: Neural network training chart for the first test.

Neural Network Training: 3 Classes				
Total Hidden Nodes	Fit Tester Error	Perc. Error Normal State	Perc. Error Disabled Node	Perc. Error Excess User Throughput
15	0.29	0.09	0.10	0.08
24	0.24	0.06	0.07	0.05
30	0.22	0.04	0.05	0.04
33	0.19	0.03	0.04	0.03

Table 4.2: Neural network training chart for the second test.

Neural Network Training: 5 Classes						
Total Hidden Nodes	Fit Tester Error	Perc. Error Normal State	Perc. Error Disabled Node	Perc. Error Excess User Throughput	Perc. Error Degraded Buffer	Perc. Error Disabled Link
175	0.20	0.13	0.16	0.14	0.19	0.21
200	0.15	0.10	0.13	0.13	0.17	0.18
210	0.14	0.08	0.11	0.12	0.16	0.15
230	0.13	0.07	0.10	0.09	0.14	0.11

Table 4.3: Neural network training chart for the third test.

In the fourth and final test, we again considered all five classes of faults and trained the RBFNs with different sample sizes. We trained the RBFNs with 180 samples each for the normal, disabled node, and excess user traffic classes. For the remaining two faults classes, we used 320 samples for each class, giving a total of 1180 points in the training set. The reason for this is because these two cases do not manifest themselves in an obvious manner through the performance data from the network. When the testing for these two cases was performed with 180 samples per class, the percentage of misclassification was very high (approximately 0.40 for each fault class). On the other hand, when we tried using 500 samples for each of these two classes, the RBFN was "overtrained" and all data points (from the testing data) were classified either as "degraded buffer" or "link failed". Thus, we had to use an intermediate number of points between these two extreme cases of training and the results for this case are shown in Table 4.4.

Neural Network Training: 5 Classes						
Total Hidden Nodes	Fit Tester Error	Perc. Error Normal State	Perc. Error Disabled Node	Perc. Error Excess User Throughput	Perc. Error Degraded Buffer	Perc. Error Disabled Link
200	0.30	0.12	0.35	0.55	0.35	0.10
250	0.26	0.04	0.44	0.33	0.52	0.10
300	0.20	0.33	0.50	0.20	0.31	0.10

Table 4.4: Neural network training chart for the fourth test.

Another interesting observation was noted when the neural networks were tested for detecting faults under the following scenario. Consider a neural network that was trained to detect disabled nodes using data when Los Angeles and Utah were disabled independently, i.e. they were not disabled simultaneously. The network is shown in Figure 4.12. When either one of these nodes is disabled during an actual simulation, the neural network will detect the fault, depending on how well it was trained. However, if testing data is presented to the RBFN with the node at Phoenix being disabled, the RBFN may not always indicate the correct type of fault, i.e. it does not always give "node disabled" as its output. After analyzing the behavior of the network under fault conditions, it appears that the network topology is the main reason for these observations. Since all occurrences of a particular fault class are not identical, i.e. the behavior of the network (re-routing of packets, etc.) due to a fault in Los Angeles is not identical to the behavior of the network due to a fault in Utah, several different cases need to be presented to the RBFN for the same fault class. Obviously, this corre-

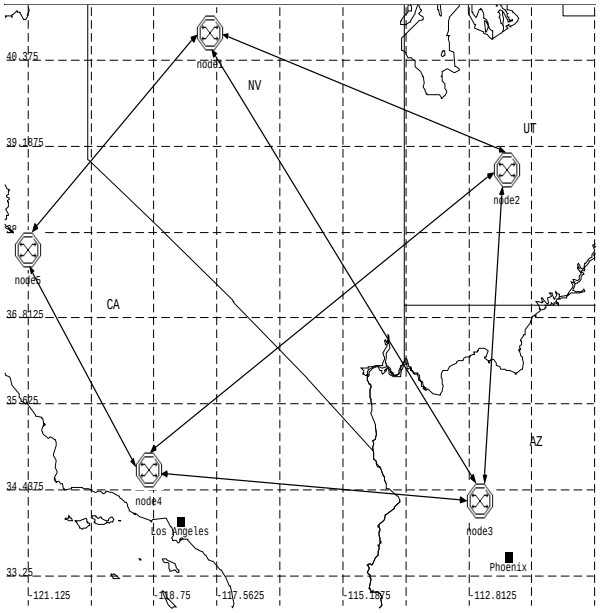

Figure 4.12: Diagram illustrating a typical subnetwork.

sponds to longer training sessions for the neural networks. Similar observations were also recorded for the other fault classes.

The output of the neural network is used by a classifier to inform the network operator of the current status of the network. The codes used in our implementation are listed in Table 4.5. If a certain fault code (other than 0) is observed several times (e.g. K times out of M samples, where K and M are defined by the network designer), then the expert system is activated to determine further information about the location and cause of the fault, as described in the next section.

4.4.2 Second Level of Diagnosis: Expert Systems

The neural network for each subnetwork analyzes the incoming data and if a state other than a normal one appears to be present, then the expert system makes queries to an ORACLE database to determine further information about the

Fault Codes for the Neural Network Classifier	
Network State	Fault Code
Normal	0
Node Disabled	1
Excess User Traffic	2
Reduced Switch Capacity	3
Link Down	4

Table 4.5: Network states and their respective fault codes for the neural network classifier.

observed fault in the network. Different fault conditions have different queries, as described below.

1. When a *node failure* occurs, we need to look for a SNMP trap. In addition, the following queries can be made to confirm the reception of the trap:

   - The utilization levels of each of the outbound links from the failed node should be less than some number $\epsilon > 0$.
   - Examine x25StatCallTimeouts counter (at the source DTEs). It should increase. By identifying a source/destination pair, as it is possible for a permanent virtual circuit (PVC), the location of the problem can be traced.
   - Look at x25StatOutCallFailures counter at the source DTE. It should increase. This can be checked in conjunction with the x25StatOutCallAttempts

parameter.

- Look at the x25StatRestartTimeouts and x25StatResetTimeouts counters. They should increase.

2. When *excess traffic is generated by a user*, causing network congestion to occur, we need to look for an alarm indicating this condition. This alarm occurs as a result of an increase in measured packet throughput at the node (in the X.25 cloud) connected to the source DTE generating the high traffic. In addition, we also need to look at the following:

   - The value of the x25StatOutDataPackets parameter for the DTEs connected to the node where the high throughput measurements were made.
   - The value of the x25StatInDataPackets parameter for the destination DTE in the case of a PVC.
   - The value of the x25StatInCalls for the destination DTE in the case of a PVC. Although this is a useful parameter, it should be used in conjunction with the other parameters listed above and not used alone. This is to allow for the possibility of there being a small number of calls to the named destination, with each call having a lot of traffic (i.e. a high value of x25StatInDataPackets).

3. For a *degradation in the service rate of an input queue*, a SNMP trap would be generated. In addition, the following queries can be made to the database:

   - An alarm reflecting high queue sizes and possibly blocking.

- An increase in the end to end delay experienced by packets.

4. When a *link failure* occurs, a SNMP trap will be generated. In addition, the following query can be made to confirm the reception of the trap:

   - A link that has zero utilization in both directions (recall that the links in the X.25 cloud are bidirectional).
   - Look at the x25StatRestartTimeouts and x25StatResetTimeouts counters. They should increase.

   All the points mentioned above agree with the simulation results from OPNET. These points are summarized in the next chapter in a step-by-step format.

Chapter 5

Fault Detection Algorithm

5.1 Single Faults

- To detect a node failure at node i, the first thing to look for is a SNMP trap. Reception of a trap would solve the problem. If, due to some problem in the network, the trap was not received by the SNMP manager (a feature that exists in our simulation), then we execute a query from the expert system looking for the following condition:

$$\sum_{\forall j s.t. \exists link(i,j)} \rho_{ij} < \epsilon$$

In theory, we would require the sum to be equal to zero. However, in a software implementation, we would set ϵ to be a very small number, such as 0.01, and changing the equality to an inequality as shown above.

To confirm the hypothesis, examine:

1. x25StatCallTimeouts counter at the DTEs that are the "source" part of the source/destination pairs for the DTEs connected to node i.

2. x25StatOutCallFailures and x25StatOutCallAttempts counters at the source DTE.

3. x25StatRestartTimeouts and x25StatResetTimeouts counters.

- To detect a user connected to node i that is submitting excess traffic to the network, we look for the following condition:

$$\sum_{\forall j s.t. \exists link(i,j)} \rho_{ij} > \tau$$

To confirm the hypothesis, check the following:

1. x25StatOutDataPackets at the DTEs connected to node i.

2. Measured packet throughput at node i.

3. x25StatInDataPackets at the destination DTE, i.e. node i, obtained by checking the source/destination pairs in the case of PVCs.

4. x25StatInCalls at the destination DTE, obtained by checking the source/destination pairs in the case of PVCs.

- To detect a degraded switch, start by looking for a SNMP trap. Reception of a trap would solve the problem. To confirm the hypothesis, check the following:

1. Alarms corresponding to high queue sizes and/or blocking of packets.

2. High end to end delays experienced by packets.

- To detect a link failure on link (i, j), i.e. the link connecting nodes i and j, the first thing to look for is a SNMP trap. Reception of a trap would

solve the problem. If the SNMP manager does not receive a trap, then we execute a query from the expert system looking for the following condition: find i and j such that

$$\rho_{ij} = 0$$

and

$$\rho_{ji} = 0$$

In addition, look at the x25StatRestartTimeouts and x25StatResetTimeouts counters.

5.2 Multiple Faults

In the case of multiple faults, we simply need to look at the outputs of the RBFN neural networks and see which ones do not correspond to normal traffic. By doing so, we eliminate a large number of nodes in the X.25 network and we can focus on those subnetworks that are experiencing problems. In this project, we are not considering multiple faults occurring simultaneously within the same subnetwork since the probability of occurrence of such an event is much smaller than the probability of occurrence of multiple faults within different subnetworks.

Chapter 6

Conclusion

In this project, we have explored the area of automated fault management of data networks. Rather than taking a purely reactive approach towards this problem, we have used a proactive *and* reactive approach. The former is present through the minimum cost routing algorithm in the X.25 network. The latter is present through the use of artificial intelligence techniques. We used these techniques for performing fault detection and diagnosis, using performance data, SNMP network statistics, alarms, and SNMP traps. Although the OPNET simulation consisted of a relatively small number of nodes, that is not a problem since we have divided our X.25 network into a group of subnetworks and assigned a RBFN network to each of these subnetworks. The idea of using one neural network for each subnetwork (in the X.25 network) is appealing because it leads to a *scalable* solution in the case of large networks. For example, if a certain company has 100 subnetworks (which implies 100 RBFN neural networks) and they decide to expand their network by adding another subnetwork, then they would only need to train one more RBFN network without having to change the other 100 RBFN networks. Furthermore, the expert system queries are common

to all subnetworks and can be used with minimal modification, if any. This leads to code reuse, an important element of any software development project. Although it takes time to train the neural networks (since there is no unique training scheme that can be used), once a certain performance (or a desired error criterion) is obtained, the neural network can be used with high confidence, as long as the network topology of the subnetwork (in the X.25 network) has not changed. This is not an unreasonable assumption since many companies choose several of the traditional network architectures (such as token bus, token ring, star, etc. Thus, the time spent initially (to train the neural networks) leads to long term dividends from a network management point of view.

In terms of future work in this area, some possibilites include working with other protocols, such as TCP/IP, for example. Alarm correlation is also a very important issue in this area. Correlations can be drawn between the alarms and perhaps the SNMP traps.