THE DESIGN AND EVALUATION OF NETWORK SERVICES IN AN ACTIVE NETWORK ARCHITECTURAL FRAMEWORK

by

NIRAJ PRABHAVALKAR

A thesis submitted to the

Graduate School-New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

And approved by

Professor Manish Parashar

_____

_____

_____

_____

New Brunswick, New Jersey

October, 200

ABSTRACT OF THE THESIS

The design and evaluation of network services in an active network architectural framework

by NIRAJ PRABHAVALKAR

Thesis Director:

Professor Manish Parashar

There are an increasing number of applications that require more support from the network nodes besides the storage and forwarding of bits that the nodes presently provide. These applications include group communication strategies, scalable network management, provisioning for quality of service, efficient routing protocols and congestion control mechanisms. Active networks provide a new networking platform that is flexible and extensible at runtime and supports the rapid evolution and deployment of networking technologies to suit current needs. They allow the network nodes to perform application specific computation on the data flowing through them. Although, with active networking the possibilities for refining current applications and introducing new ones are tremendous, it is important to demonstrate the performance benefits accrued from an active networking platform.

Despite research efforts in industry and academia to eliminate network congestion, the problem continues to persist. Furthermore, a number of applications require a constant bit rate of transmission while some others tend to ‘grab’ as much network bandwidth as available ignoring congestion related feedback from the network. We utilize the processing capabilities of active networks in order to effectively control bandwidth greedy connections at a congested node.

Traceroute is a popular network utility that discovers the route followed by an IP datagram to another host. Refinements in accuracy of operation and savings in network resources are achieved by using an active networking platform to implement traceroute.

This thesis investigates the design of an experimental active network testbed and develops active services that utilize the underlying network fabric. It makes the following contributions.

- The design and implementation of an active network testbed comprising of interconnected active nodes using object-oriented techniques. In our network model datagrams may select specific processing at the active nodes from an available set of options thus conforming to a menu-driven approach.
- We have designed and evaluated a congestion control mechanism that aims to limit the degradation in network performance caused by bandwidth greedy applications. The mechanism operates by monitoring packet queues to detect a greedy connection. A process of recursive mobile filtering then controls the identified connection. Specifically, we install a packet filter for the greedy connection and use active messages to dynamically move the filter towards the source of the connection. By filtering packets closer to the source, the network resources are protected from the aggressive flow.
- We have implemented an active traceroute utility that achieves considerable savings in time complexity and link utilization for achieving the same objectives as traditional traceroute.

Table of Contents

List of Illustrations

Chapter 1

Introduction

An active network may be simplistically viewed as a set of "Active nodes" that perform customized operations on the data flowing through them. Traditional data networks were designed with the aim of transferring bits from one end system to another. The transport mechanism achieved its objectives with minimal computation within the network. In contrast, active networks allow the network nodes to perform computation on the data passing through them. In fact, some implementations also allow their users to inject customized programs into the nodes of the network that may modify, store or redirect the user data flowing through the network.

An active network is a relatively new concept gaining popularity in 1996. The active networks program has the goal of producing a new networking platform that is flexible and extensible at runtime. This platform aims to support the rapid evolution and deployment of networking technologies to suit current needs and also help in developing services such as group communication strategies, scalable network management, quality of service, efficient routing protocols and congestion control mechanisms. The active network architecture supports a finely tuned degree of control over network services. The packet itself is the basis for describing, provisioning, or tailoring resources to achieve the delivery and management requirements. One such architecture makes use of a “Smart Packet” [21] as the basic message unit on the network. This packet is an agent with its objectives expressed through a portion of the packet called its "method" -- a set of instructions that can be interpreted consistently by the active nodes through which it traverses. The network is engineered to allow security, reliability, availability and quality of service to be tuned at multiple levels of granularity under a wide range of conditions. The active networks program involves the synthesis of work in programming languages, operating systems and computer networking. Figure 1.1 shows a comparison of network processing

between traditional networks and active networks. It has been taken from the official DARPA website [20].

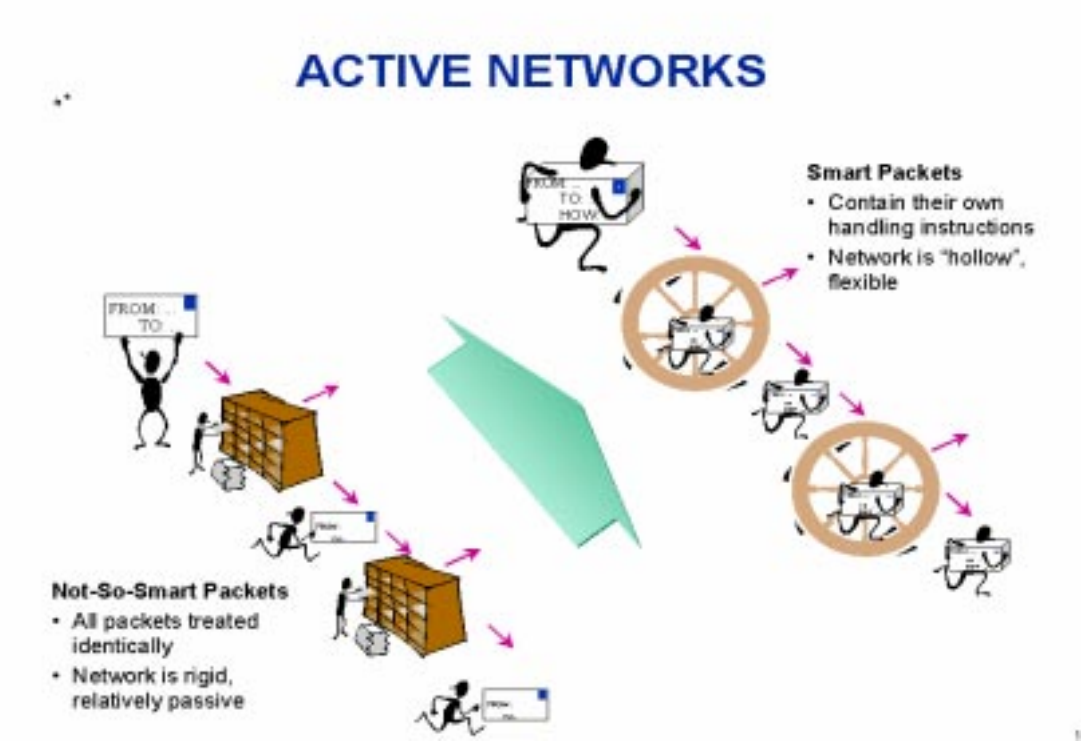

Figure 1.1 Comparing traditional networks with active networks

The objectives of this thesis are:

- To provide a 'proof of concept' for active network technologies. In doing so we wish to develop an active network testbed, design applications that utilize the added functionality of the testbed and present relevant results obtained.
- To design and build an active network testbed for interconnecting lightweight active nodes. The testbed must be extensible, user-friendly and should efficiently accommodate traditional forwarding services.
- To implement new network utilities or improve upon existing ones by using the processing capability of intermediate nodes and to demonstrate the effectiveness of the utilities by experimental evaluation.

1.1 Background and motivation

The fundamentals for introducing a novel computer networking architecture must be sound. In the following sections we outline the motivation for developing active network architectures.

1.1.1 A lengthy process called 'Standardization'

Traditional networking architectures evolve at a slow pace governed by the time taken for standardization and deployment of new protocols. We need to match this evolution speed with the speed at which new applications are being introduced into networking. The design philosophy of TCP/IP networks is based on a layered approach with each layer communicating with its peer using standardized protocols. A wide variety of high level services such as file transfer (FTP), E-mail (SMTP, POP) and Hyper-text transfer (HTTP), and low level network technologies such as ATM, FDDI and Ethernet can be made to inter-operate at the network level by funneling their functionality's through the static IP protocol. Thus, IP routers are configured using a hardware approach with the fixed IP protocol format in mind leading to the hour-glass model of TCP/IP networks as shown below.

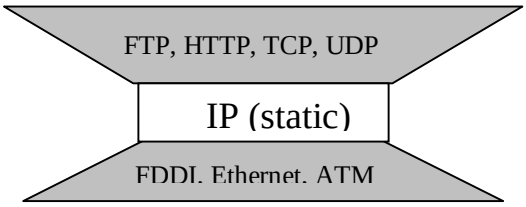

Figure 1.2 Hourglass model of TCP/IP networks

As the Internet grows it is increasingly difficult to maintain, let alone accelerate the pace of innovation [6]. Every time a sophisticated application emerges or a change in link layer network technology occurs we need to standardize and deploy new protocols in order to conform to the interoperable IP layer. Standardization and deployment of such protocols is a lengthy and time-consuming process taking several years as RSVP (Resource Reservation Protocol) and IGMP (Internet Group Management Protocol) have proved. By having programmable open nodes and the ability to deploy programs dynamically into the node engines, network services are decoupled

from the underlying hardware. This allows new services to be demand loaded into the infrastructure. Instead of hard-coding the functions of the network nodes, the execution environments deployed for the application specific programs needs to be agreed upon so that innovative ideas can be rapidly inducted over the underlying substrate.

1.1.2 Application Support

There are an increasing number of applications that require more support from the network nodes besides the storage and forwarding of bits that they presently provide. Some of these applications are listed below.

- The World Wide Web has a client-server design with clients establishing connections with servers and requesting data from them. Caching "popular" data close to "interested" parties reduces the latency of the data transfer and also reduces load on the server(s). However, deploying caches dynamically at strategic locations within the network is a non-trivial task and cannot be supported without the development of new protocols.
- Multicasting takes place with the help of routers having added functionality to route IP packets to multiple destinations.
- Mobility of hosts connected to the Internet requires the presence of mobile proxies in the network that can re-route traffic to the correct location of the host.
- Multimedia applications like video require transcoding mechanisms at strategic locations within the network to convert high bit-rate streams to lower ones. The transcoder is based on some data characteristics such as resolution, frame rate, etc.
- Installing firewalls in the network at administrative boundaries provides intranet security. The firewalls are essentially filters that examine transit traffic and allow only conforming traffic to pass through while blocking other traffic. The manual process of updating firewalls to enable the use of new applications is an impediment to the adoption of new technology and needs to be automated.

- A Network utility such as traceroute allows users to discover the route of an IP datagram from a one node to another. The utility assumes static routes for consecutive datagrams injected by the source node. We have investigated the impact of active networking technologies on optimizing the performance of the traceroute utility in terms of reducing complexity and eliminating the static route assumption.
- Many congestion avoidance mechanisms rely on routers and end-hosts to control connections responsible for causing congestion so as to prevent further degradation of the network. However, controlling 'bandwidth greedy connections[1]' continues to remain an open problem in computer communications. In this thesis we will investigate a congestion control strategy that is aimed at detecting and isolating such connections using active networks.

All the applications described above require some enhanced capabilities within the network to achieve successful operation. In the absence of architecture support, the present solution consists of a collection of ad-hoc approaches like installing Web proxies, multicast routers, mobile proxies, video gateways and firewalls to provide the above services to end-users. The obvious questions are "Can we have a more generic solution to support a variety of applications, some of which may not even exist as of today?" and if so "What is the appropriate approach?"

1.1.3 Technological progress

Computationally powerful machines are readily available as desktop PC's today. We may safely assume that in the coming years processors will become smaller and faster. The same applies to network processors such as routers. Advancement in technology will help in developing a generic network model capable of performing customized computation within the network and not restricted to the end points as is done today.

1.2 The challenges

The world of computers and communications distinguishes between nodes used for computing and nodes used for communications [35]. This distinction evolved naturally since computers were

developed as stand alone machines that were subsequently connected by network elements. An active network tends to narrow the gap between intermediate nodes and end-hosts with the introduction of programmable open nodes that 'may' be injected with code from end-hosts. Figure 1.3 illustrates a generic active node model and is based on the network model presented in [14]. It comprises of three principal components; a forwarding engine for storing and forwarding packets in the network, a transient execution environment for application oriented packet processing and an accessible storage location for the execution environment. Based on this model the most notable technical challenges in making the transition from the present Internet to an active network are network security, evaluating performance benefits, addressing interoperability and deployment.

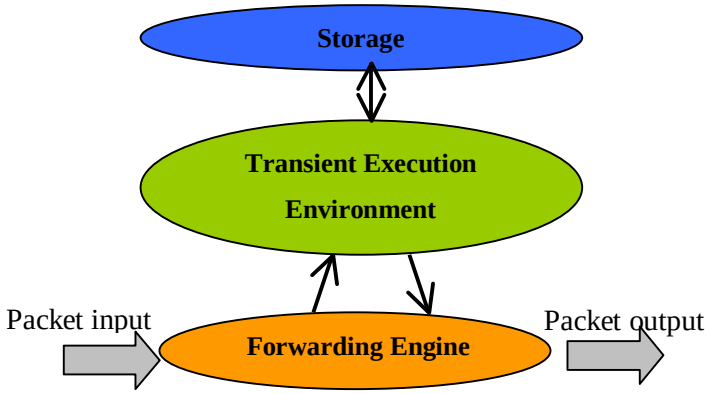

Figure 1.3 Generic active node model

1.2.1 Network security

Security issues are critical in active networks especially in implementations that let users load their own code. The origin of information needs to be authenticated and it must be protected from modification. Active code may have access to network resources making active networks particularly susceptible to malicious or defective code that could threaten the operation of the network. Researchers are finding ways to build networks in a way that will pre-empt defective programs from harming the network or interfering with other programs and other users. But this

[1] Bandwidth greedy connections are defined in chapter 4.

approach also leads to a dilemma since we need to carefully restrict the actions of arbitrary code while providing that very code with the flexibility of network level primitives.

1.2.2 Performance evaluation

The overhead of processing active packets at intermediate nodes in the network makes them fall behind traditional data networks in terms of latency and throughput. The computation required within the network may even tend to clog routers leading to network congestion. However, processing of packets is not needed at every intermediate node within the network since most applications require specific processing only at strategic locations within the network. Contrary to popular belief, despite increasing the amount of processing performed within the network, applications can improve overall system performance. Although throughput (a common network performance measure) may suffer due to processing overheads in active networks, the application may benefit on the whole if fewer active packets are needed to achieve the same application objectives. For example research presented in [37] analyzes the performance of a real time auction application that uses caching within the network backbone to reduce the load on the auction server and backbone routers in terms of server load, round trip processing time and network bandwidth consumption. During periods of heavy load the auction server activates filters within the network and periodically updates them with the current price of the popular items. The filtering active nodes are then authorized to reject bids that are lower than the current price. This active networking protocol helps in distributing the server load, reducing bandwidth consumption and cutting down on round-trip response time to customers during busy periods. Thus, some researchers point out that performance should be evaluated in terms of application specific metrics, which may not be positively correlated with network metrics. The cost of these performance improvements in Active Networking is in the increased consumption of computational and storage resources in the network, which may slow down other network traffic flowing through the busy active node. However, the competing traffic could also benefit from

active processing due to the overall reduction in bandwidth utilization and congestion-related loss.

1.2.3 Interoperability

Packet networks achieve interoperability by standardizing the syntax and semantics of packets. Internet routers support the agreed IP specifications and perform the same computation on every packet. In contrast active nodes can perform different computations on the packets flowing through them. Interoperability must be achieved at a higher level of abstraction. Rather than fixing the computation performed on the active packet we need to standardize the computational model consisting of the instruction set and resources available to the active packets. For example, to achieve cross-platform compatibility, a standard API could be developed to act as a common programming model for writing the code that is injected into active networks. This would make it easier to develop new applications as desired and would also reduce the program content of the active packet.

1.2.4 Deployment

Deploying a new system needs substantial justification along with backward compatibility. To succeed in the marketplace, proponents must develop applications, both current and future, that demonstrate a clear advantage as promised without rendering prior networking equipment useless. In order to strengthen the justification of the active networks program it is required to demonstrate the capabilities of middleware services[2] by developing suitable applications. In this work we concentrate our efforts in the development of network services on an experimental testbed.

1.3 Contributions

- We have built an active network testbed comprising of interconnected active nodes. By using object-oriented techniques in the design of our active node we provide extensibility, re-

2 Middleware services are those that manage system resources, describe message format and support data transformation.

usability and user-friendliness. Further, by creating a separate processing track for active datagrams, we have ensured minimal impact on applications requiring plain old forwarding service.

- The network utilities we have developed include (a) APing for discovering if a node in the network is operational and (b) Atraceroute for tracing the path of a datagram in the RANI active network. We have highlighted the advantages of Atraceroute by testing it on our network and comparing it with the existing implementation on traditional networks.
- We have addressed the problem of limiting the impact of 'bandwidth greedy connections' at congested nodes. We have designed and evaluated an intelligent congestion control mechanism to detect and counter such connections during periods of severe congestion. In our implementation we use a packet filter to discard packets belonging to the identified connection at the congested node and dynamically move the filter towards the source of the 'greedy' connection thus protecting network resources from being overwhelmed. We successfully demonstrated the utility of our mobile filtering mechanism through experimentation.

1.4 Thesis Overview

Chapter 2 covers related work in which we survey some of the prevalent active network architectures developed by others. It also includes a comparison of the various active network architectures. In Chapter 3, we present our active network system architecture highlighting the goals we have achieved. In Chapter 4 we present the active network services that we have developed. The chapter begins with a description of our active traceroute and active ping utilities. Finally, we examine the inadequacies of a popular congestion control strategy with respect to bandwidth greedy connections and then explain a mobile filtering mechanism for isolating such connections during periods of high congestion in our active network testbed. Chapter 5 is dedicated to experimental evaluation of our active filtering mechanism. Chapter 6 draws conclusions on the effectiveness of our work, suggesting some possible future directions.

Chapter 2

Related Work

With active networking, the network is no longer viewed as a passive mover of bits, but rather as a more general computation engine: information injected into the network may be modified, stored or redirected as it is being transported. Obviously such a capability opens up many exciting possibilities. However, it also raises a number of issues including security, interoperability and migration strategy. All of these are influenced in large part by the active networking architecture that defines the interface between the user and the capabilities provided by the network. The networking architecture adopted has a direct bearing on the utilities and applications that it may support.

We have developed a simple active network testbed. We utilize the functionality of this testbed by implementing network utilities and by designing an intelligent congestion control mechanism. This chapter is devoted to giving a brief overview of the prevailing active network architectural models. Finally, we compare these architectures and provide the motivation for the design of our testbed.

2.1 Related Active Network Architectures

2.1.1 Smart Packets

BBN is developing a capability for packets to carry programs that are executed at each node the packet visits in the network [3]. The programs implement extended diagnostic functionality in the network. The Smart Packets architecture has the following goals:

- Providing a specification for smart packet formats and their encapsulation into some network data delivery service.
- Specification of a high level language, its assembly language, and a compressed encoding mechanism for representing the portion of a smart packet that gets executed.

- Developing a virtual machine resident in each networking element to provide context for executing the program within the smart packet.
- Developing a secure design

The Smart Packets project is designed to demonstrate that network management is a fruitful target for exploiting active network technology. Making the programmable environment too rich or flexible would overload the computing power of the managed node and compromise on security. To balance flexibility with computing power and security two important decisions were made. Firstly, there would be no persistent state in the network nodes. Consequently, the programs carried by the smart packets must be completely self-contained. Even fragmentation of the smart packet is not permitted. So the programming language used must be able to express meaningful programs in a short (1Kbyte) length. Secondly, the operating environment must be secure. Also, the programming language used should avoid dangerous or superfluous features like file system access or memory management. This goal suggests that the code should be executed within a virtual machine where only controlled operations are permitted.

2.1.2 Active Node Transfer System (ANTS)

ANTS [8] is part of a continuing research effort of the Software Devices and Systems group at the MIT Laboratory of Computer Science. An ANTS based network consists of an interconnected group of nodes that may be connected across the local or wide area by point-to-point or shared medium channels [7, 37]. In addition to providing IP-style routing and forwarding as the default network-level service, ANTS allows applications to introduce new protocols into the network. Applications specify the routines to be executed at the active network nodes that forward their messages. The various components of the ANTS architecture are presented below.

Protocols and Capsules: The packets found in traditional networks are replaced by capsules that refer to the processing to be performed on their behalf at active nodes. Capsule types that share information within the network are grouped into protocols; a protocol provides a service and is the unit of network customization and protection. The most important function of the capsule

format is to contain an identifier for a protocol and forwarding routine within that protocol. The identifier is based on a fingerprint of the protocol code. Some forwarding routines are "well-known" in that they are guaranteed to be available at every active node. Other routines may be "application-specific". Typically, they will not reside at every node, but must be transferred to a node before the first capsules of that type can be processed. Subsequently, capsules belonging to a particular protocol contain the same identifier and are processed similarly at the active nodes.

Active Nodes: The active nodes execute the capsules of a protocol and maintain protocol state replace selected routers within the Internet and at participating end nodes. Unlike ordinary routers, active nodes provide an API for capsule processing routines, and execute those routines safely by using operating system and language techniques. A major difficulty in designing programmable networks is to allow nodes to execute user defined programs while preventing unwanted interactions. The ANTS approach has been to execute protocols within a restricted environment that limits their access to shared resources. The primitives of the active nodes are -

- *Environment access;* to query the node location, state of links, routing tables, local time and so forth;
- *Capsule manipulation;* with access to both header fields and payload;
- *Control operation;* to allow capsules to create other capsules and forward, suspend or discard themselves;
- *Storage;* to manipulate a soft-store of application defined objects that are held for a short interval.

The capsule format includes a resource limit that functions as a generalized TTL (Time-To-Live) field. This limit is carried with the capsule and is decremented by nodes as resources are consumed. Only active nodes may alter this field, and nodes discard capsules when their limit reaches zero.

Code Distribution Scheme: In ANTS an explicit code distribution mechanism ensures that capsule processing routines are automatically and dynamically transferred to the active nodes

where they are needed. This component does not exist in traditional networks and is handled by the system, not the service programmer. The ANTS implementation couples the transfer of code with the transfer of data as an in-band function. This approach limits the distribution of code to where it is needed, while adapting to node and connectivity failures. The code distribution scheme is suited to flows, i.e., sequences of capsules that follow the same path and require the same processing.

2.1.3 SwitchWare

SwitchWare [11, 13, 16] is an active networks research effort undertaken at the Penn Department of Computer and Information Science and Bellcore [18]. Active Networks must balance the flexibility of a programmable network infrastructure against the safety and security requirements inherent in sharing that infrastructure. The SwitchWare active network achieves this balance using three layers, each having a separate language specification. The *switchlet* language is the language with which users can access the programmable features of the SwitchWare switch. The *wire* language is the form in which the switchlets are moved between switches and the *infrastructure* language programs the SwitchWare switch. An analogy of a three level language might be a Java program written by a user, its byte code form, and the C language programs that comprise the byte code interpreter.

Components of SwitchWare include active packets, their extensions and the secure active router infrastructure. These are explained below.

Active Packets: An active packet is one that contains both code and data needed to process the packet in the network. They replace the traditional network packet with a mobile program. The code part of an active packet provides the control functions of a traditional packet header, but does so much more flexibly, since it can interact with the environment of the router in a more complex and customizable way. Similarly, the data in the active packet program replaces the payload of a traditional packet, but provides a customizable structure that can be used by the program. Basic data transport can be implemented with code that takes the destination address

part of its data, looks up the next hop in a routing table, and then forwards the entire packet to the next hop. At the destination the code delivers the payload part of the data to the application.

Active Extensions: An active extension is some code that may be loaded on a running switch to alter the processing of future packets. Node-resident extensions form the middle layer of the architecture. They can be dynamically loaded active extensions or they can be part of the base functionality of the router. They are not mobile – to communicate with other routers they use active packets. Thus extensions are base functionality or are dynamic additions rather than "mobile code". If the code can only be loaded from a local persistent store, then the extension is referred to as a local extension. Extensions may make use of other extensions already loaded on the node; they need not be independent. Extensions reside on the node, e.g., in memory or on local disk, until they are loaded. Because they are invoked only when needed, there is no inherent need for the extensions to be lightweight. The key difference between active packets and extensions is that although extensions may be dynamically loaded across the network, they execute entirely on a particular node where as active packets are executed at some or all of the active nodes it passes through.

Secure Active Router Infrastructure: This is the lowest layer of the architecture. While the top two layers emphasize support for several forms of dynamic flexibility, the lowest layer is primarily static. The goal of this layer is to provide a secure foundation upon which the other two layers build. The importance of this is clear, since no matter how much security is assured by the upper layers, security will be compromised if this layer creates an insecure environment.

2.1.4 CANES (Composable Active Network Elements)

CANES [17] is a research project at Georgia Institute of Technology. In this design, users can select from an available set of functions to be computed on their data, and can supply parameters as input to those computations. The available functions are chosen and implemented by the network service provider, and support specific services; thus users are able to influence the computation of a selected function, but cannot define arbitrary functions to be computed [31].

This approach has some benefits with respect to incremental deployment as well as security and efficiency: Active Network functions can be individually implemented and thoroughly tested by the service provider before deployment, and new functions can be added as they are developed.

2.1.4.1 Architecture Overview

The CANES architectural model for active networks takes a menu-based approach in which the active node supports a fixed set of active functions and the active packets indicate the function(s) to be invoked and supply parameters to those functions. The basic idea behind this architecture is the incremental addition of user-controllable functions, where each function is precisely defined and supports a specific service. The function specifications include:

- The *identifier* associated with the function.
- The *parameters* associated with the function and the method of encoding them in the packet.
- The *semantics* of the function. Ideally, the function semantics would be given in a standard notation or another notation developed specifically for the purpose. A standard environment, comprising support services such as private state storage and retrieval, access to shared state information (e.g. routing tables), message forwarding primitives, etc., would provide a foundation on which new services could be built.

CANES delegates the addition of a new function to a network node to the network service provider. As with current networks, once a function is specified, each provider or vendor would be free to implement the functionality in a manner consistent with the specification. This approach corresponds roughly to the way new features are deployed in the public switched telephone network today; users have the option of selecting from a variety of features implemented by the service provider.

2.1.5 NetScript: A language and Environment for Programmable Networks

NetScript [19] is a programming language and environment for building networked systems. Its programs are organized as mobile agents that are dispatched to remote systems and executed under local or remote control. The goal of NetScript is to simplify the development of networked

systems and to enable their remote programming. NetScript could be used to build packet stream filters, routers, packet analyzers and multimedia stream processors.

2.1.5.1 The NetScript Network and its target applications

A NetScript network consists of a collection of network nodes (e.g. PCs, switches, routers) each of which runs one or more NetScript engines. The engine is a software abstraction of a programmable packet-processing device. Each NetScript engine consists of dataflow components, called boxes, that process packet-streams that flow through them. Packets flowing through a NetScript node are processed by successive boxes to perform protocol functions. The system consists of two components: NetScript, a textual dataflow language for composing packet-processing protocols and the NetScript Toolkit, a set of Java classes to which the textual language compiles. The boxes form a reactive system in which data (in the form of packets) flows from one box to another. Arrival of data at one or more input ports of a box triggers computation within that box; otherwise the box sleeps until data arrives to trigger it. The box is the central construct in NetScript and the unit of program composition. A box declaration consists of four parts: the box name, input port and output port declarations, a declaration of internal boxes and a connect statement that defines the connections between internal boxes. When a box is loaded at a NetScript engine, NetScript will instantiate its internal contents and make connections between these boxes. Typical NetScript boxes do packet header analysis, packet demultipexing, or other protocol functions. The boxes can be dispatched to remote network engines and dynamically connected to other boxes that reside there to extend the network with new communication functions. For example, an IP router implemented in NetScript could be dynamically extended with firewall functions. Such a router might also be extended to monitor traffic, support content filtering on the edge of a network domain, or perform load balancing and traffic shaping. NetScript is useful in applications that process packet-streams.

A key application of NetScript includes the support for distribution of management functions. In order to manage a network, applications must monitor, analyze and control elements by

processing their instrumentation data. Other management technologies such as SNMP [23] have focused on moving data from elements to a management platform where applications processed this data. NetScript aims to complement these technologies with one that allows a management platform to dispatch programs (agents) to remote elements. Rather than bringing element data in real time to applications, applications could be dispatched to process the data right at the elements. This permits localization of management control loops in managed elements; in contrast SNMP stretches control loops across the network.

2.2 Comparison of architectures

Principally, there are two ways in which the active network can support processing at intermediate nodes in the network. In the *language-based* approach the active datagrams carry programs that are executed in a suitable environment. Users are allowed to inject code into the network making the system highly dynamic and flexible. However, special care must be taken to safeguard the system against malicious users and buggy code. In the *menu-based* approach the active node supports a fixed set of services. Designated operators may add new services into the node. Active datagrams carry a reference to the type of servicing they require. The implementation details of services are hidden from end user applications. We believe that the menu-based approach gives a strict administrative control over the services that the network can offer and provides a secure infrastructure at the cost of reduced dynamism. Thus, we adopt the menu-driven approach in designing our active network.

The current architectures are in the developmental phase and a consensus on a standard architecture has still not been reached. Our aim is to develop network services in an active networking environment and subsequently evaluate them. Hence, we have designed and implemented a testbed network (RANI) that is explained in Chapter 3. Table 2.1 shows a comparison of the active network architectures described in section 2.1. The list of contributions and applications is not exhaustive.

Architecture	Approach	Key Contributions	Applications
Smart Packets	Language-based	Mobile agents	Network management and diagnostics
ANTS	Language-based	Application specific protocol development	Distributed applications and web caching
SwitchWare	Language-based	Programming language development, network security	Active bridges, bootstrap architectures
CANES	Menu-based	Active components	WAN caches, selective packet treatment
NetScript	Language-based	Designing scripts, mobile agents	Management by delegation
RANI	Menu-based	Design and implementation of an active network testbed	Controlling bandwidth greedy connections, Active Traceroute

Table 2.1 Comparison of active network architectures

Chapter 3

RANI Active Network Architecture

The Internet Protocol (IP) does not support application oriented processing of datagrams at intermediate nodes. For an active datagram however the node must process the contents of the datagram (if it supports active networking) before forwarding it. This chapter describes the design and implementation of the network architecture. The network testbed is used for experimental evaluation of our network services.

3.1 Design overview of RANI (Rutgers Active Network Initiative)

The RANI network consists of a number of active nodes connected to each other via virtual links. For the sake of simplicity, we assume that the virtual links are reliable in delivering datagrams. Any node can communicate with other nodes in the network by sending datagrams across the virtual links. Datagrams that do not need active processing are referred to as passive datagrams. Passive datagrams are simply stored and forwarded similar to traditional network forwarding. Datagrams that request additional processing at the intermediate nodes in the network are called active datagrams. Active servicing is requested through a field in the header of the active packet. Each datagram is considered an atomic element and is processed individually by the active nodes.

3.1.1 Components of the RANI active node

The purpose of the active node is to service the active datagrams and to forward the passive datagrams towards their destination. Servicing active datagrams may include forwarding them. Active datagrams are serviced on a best effort basis and may result in a change in the packet's contents. We have divided the various functions of the active node into individiual modules that interoperate with each other. The Receive (Rx) and Transmit (Tx) modules handle datagram propagation issues in the network. Active datagrams are serviced in a suitable environment called the Processing (Px) module. The node resident services and programs are located in the Storage area. End users may inject active datagrams into the network and request a particular type of

service. They may also inject passive datagrams that require the traditional forwarding service. In order ensure speedy delivery of passive datagrams, we have created separate paths for active and passive datagrams as shown.

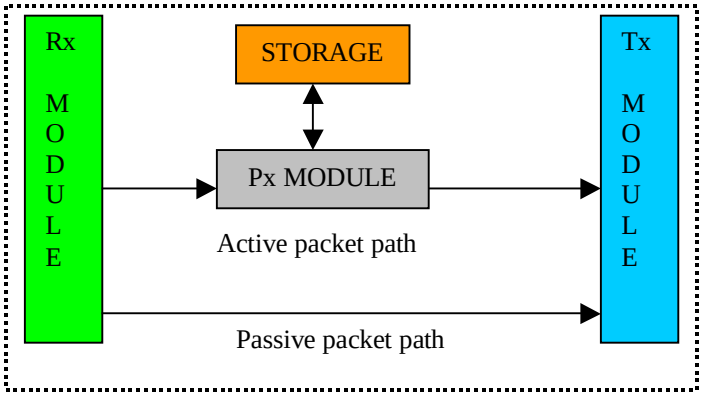

Figure 3.1 High-level design of the active node

3.1.2 Datagram propagation and 'tunneling'

We do not expect all nodes in the network to be active nodes. The virtual links that interconnect active nodes need not consist of a physical connection between the nodes. Virtual links provide a path between the two nodes that it connects. The physical path corresponding to a virtual link could traverse across legacy intermediate routers. In effect the virtual link provides a *tunnel* for transferring datagrams between active nodes.

To illustrate the use of our active network, let us consider the path of a datagram requesting service X, from source node S to destination node D. For this example, lets assume that the network nodes have been configured correctly and a virtual link between node S and node D exists. At node S, the datagram is sent to the processing module and X is executed on it. S compares its own address with the destination address of the datagram. On determining that the datagram has not reached its destination, S sends the datagram across the virtual link towards D. This action takes place at every active node along the way until it reaches D. At D the datagram is again serviced and finally delivered to the application.

3.2 Implementation Details

Our active node is implemented in Java (v1.1) as a user space process on the Windows NT operating system. The node runs at the application layer in the TCP/IP protocol stack. Application oriented processing of active packets may be required at the end nodes as well as intermediate nodes in the network. Thus we do not distinguish between intermediate nodes and end nodes.

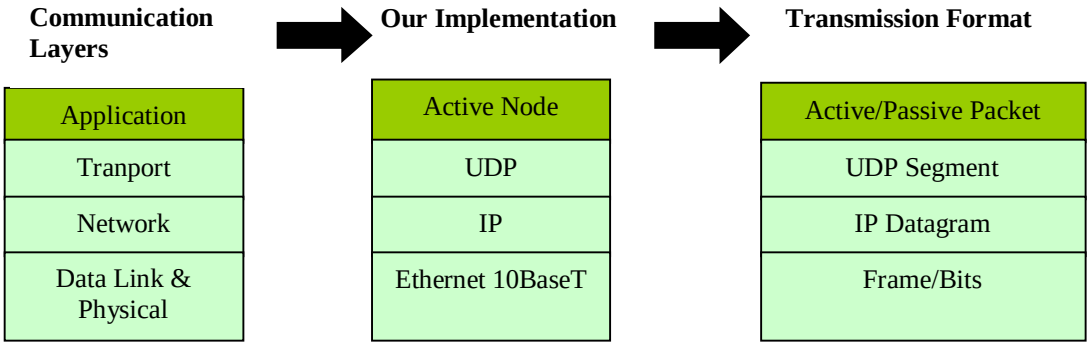

Figure 3.2 Active Node implementation

Virtual links are implemented as a UDP (User Datagram Protocol) socket pair – one socket is used for receiving datagrams and the other for sending them. Active or passive packets are created and subsequently injected into the active network via the user interface at the node. These packets are propagated as UDP segments.

3.2.1 The RANI node

The receive module comprises of UDP receive sockets for incoming datagrams and a packet filter for separating active and passive packet paths. Each receive socket contains a blocking receive thread running in an infinite loop to pick up datagrams and deliver them to the packet filter. The process module comprises of an execution engine (EE) where active packets are serviced. Active packets are serviced on a first come first served basis by ordering the packets in a FIFO execution engine queue. An independent EE thread extracts the first packet from the EE queue and dispatches it to the EE for processing. The EE thread runs in an infinite loop extracting each packet till the queue empties. The Storage (Sx) module comprises of node resident services and tables such as the routing table. The Transmission (Tx) module consists of UDP send sockets, a node queue and a single transmit thread. The node queue is common to all packets (active or

passive) that need to be forwarded. The transmit thread extracts packets from the node queue and delivers them to the next hop active node via the virtual links.

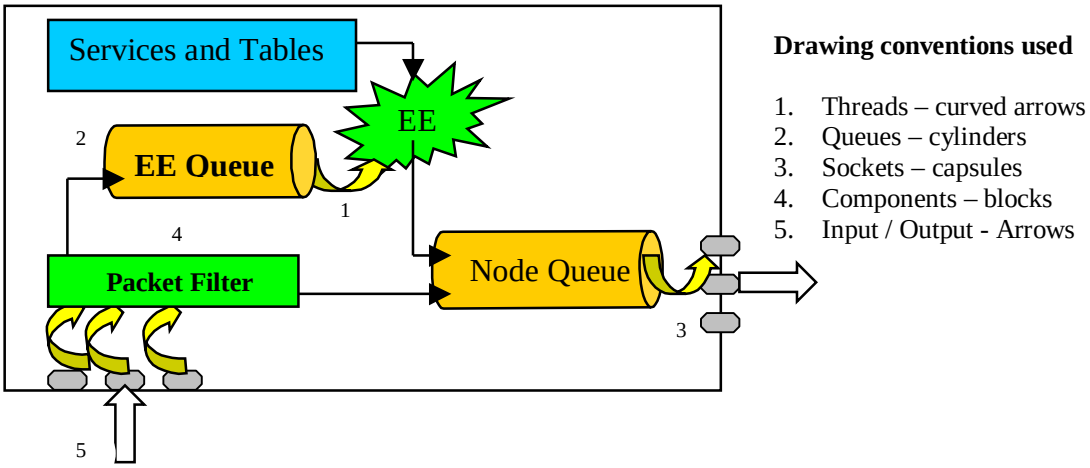

Figure 3.3 The RANI node

3.2.2 Packet format within the node

Before getting into the details of the different components as shown in the above diagram let us take a look at the packet format within the active node. Datagrams are propagated as UDP segments in byte array format across virtual links. However once inside the node, the datagram is converted into either a passive packet or an active packet.

The fields of the active packet are shown below. All the packet fields are in string format and are initially set at the source node. In comparison to the traditional datagram format, the active packet has an additional Ack, Act, PrevNode, TL and TOS fields. The packets carry state information in the TTL and PrevNode fields since these fields *must be* modified in transit by the active nodes. The Payload field *may be* modified in transit depending upon the service requested by the end-user. We have not provisioned for sequence numbering of packets since at this stage we have assumed that the network is reliable and have developed network services that deal with individual active packets.

SA	SP	DA	DP	Ack	Act	TTL	PrevNode	TL	TOS	Payload

Figure 3.4 Active packet format

SA (Source Address): It is the IPv4 address of the node that injects the packet into the network.

SP (Source Port): This field identifies the port number of the virtual link at the source node through which the packet is injected into the network.

DA (Destination Address): It is the IPv4 address of the destination node for the packet.

DP (Destination Port): This field identifies the port number of the link at the destination node on which the packet is to be received.

Ack (Acknowledgement): This field is true for acknowledgement packets and is false otherwise.

Act (Active): This field is set to true if the packet is active and is false otherwise. It distinguishes between active and passive packets.

TTL (Time To Live): This field represents an upper bound on the resources that the packet can consume within the active network. We have kept this resource bound in terms of time. The TTL field is decremented by active nodes along the way upto the destination node by the amount of time that the packet exists at the node. If a packet requires excessive processing at a node, it will reside for a longer duration at the node and correspondingly a larger value will be subtracted from its TTL resource. An intermediate node discards a packet whose TTL has dropped to zero. The TTL field is used to discard stale packets by keeping an upper bound on the time that a packet resides in the network and for calculation of packet round trip time.

PrevNode (Previous Node Visited): This field contains the IPv4 address of the node last visited by the packet and the port number of the last virtual link on which it traversed. The field is set just before an active node transmits the packet. Any node in the network can determine the previous node through which it received a packet by looking up this field. In our present implementation since we have assumed bi-directional reliable virtual links, this field is unused for applications developed so far. However, once this assumption is no longer necessarily true in future implementations, this field will be useful in developing network applications that rely on the path traversed by an active packet.

TL (TOS Length): This field carries the length in bytes of the TOS field. Keeping in mind the flexibility of introducing new services, we keep the TOS field to be of variable length.

TOS (Type of Service): The active packet requests a particular service through this field. The active node provides the service requested on a best-effort basis. For example, if a packet requests the AtraceRoute service its TOS field is set to AtraceRoute and its TL field is set to 11.

Payload: This field carries the payload of the active or passive packet.

The passive packet has the same format as the active packet with the Active field set to false and the TL and TOS fields omitted since they do not request any service from the intermediate network nodes.

3.2.3 Packet movement in the RANI network

Figure 3.5 illustrates the mechanism of injecting packets into the active network from an active node. The dark line shows the physical path that a packet traverses in our active network. From the end user application perspective the dotted arrow shows the virtual path that the packet traverses. The diagram also brings out the concept of 'tunneling' packets through legacy intermediate routers.

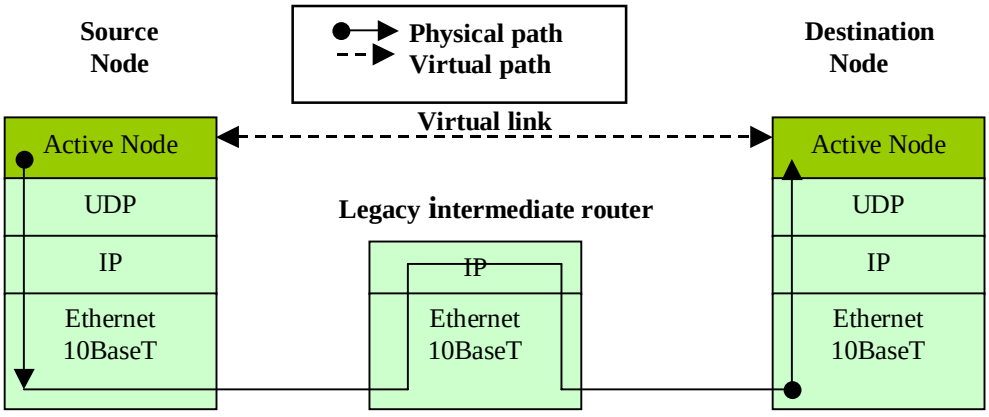

Figure 3.5 Movement of packets in the RANI network

3.2.4 The receive module

Arriving datagrams at the active node are cast into active or passive packets in the packet filter. An active packet resides at the node till it is completely serviced. Every packet in the node is subject to a destination check to ascertain if it has reached the destination node. Basically, in the

destination check, the IP address of the node is compared to the destination address field of the packet. If the fields match, the test is successful and the packet is delivered to the application. If the test is unsuccessful the packet is added to the node queue (Node Q) for forwarding. Passive packets are subjected to a destination check in the packet filter itself. Active packets are directly dispatched to the execution engine queue (EE Q) by the packet filter. The destination check for active packets is performed in the process module. By maintaining two separate queues for servicing (EE Q) and forwarding (Node Q) we create slow and fast tracks for the active and passive datagrams respectively. If we were to maintain a single queue, the passive packets would suffer from larger delays due to the longer processing time taken for active packets at the head of the queue.

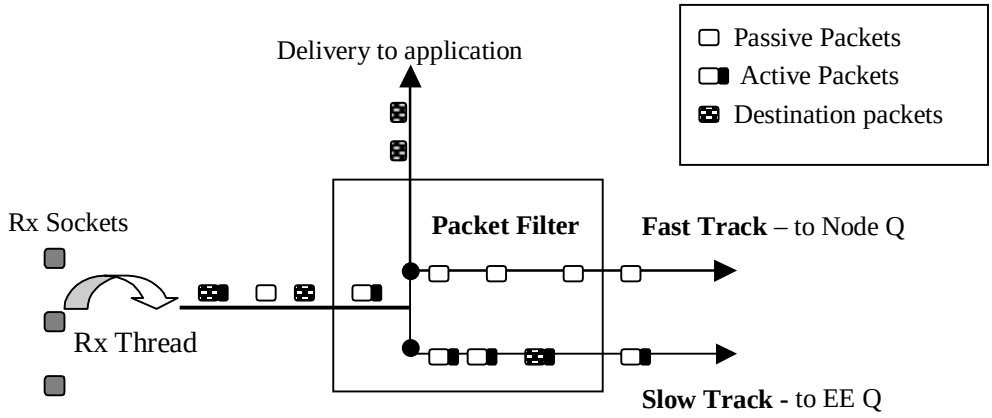

Figure 3.6 Receive (Rx) Module

Note that when an active packet reaches the destination node it is serviced before being sent to the application. Passive packets are delivered directly to the application when they reach the destination node. Figure 3.6 shows the receive module in the RANI node. Here, the spotted packets are the ones that have reached their destination.

3.2.5 The process module

Active services are stored as loadable classes in the node. They implement the *LoadableClass* interface and contain a *process* method. Active packets are serviced by invoking the *process* method of the loaded service class. We maintain a list of all services loaded at the node during its run time. This list is implemented as a hash table containing the service name as the key and the

class descriptor as the value. Packet servicing occurs in the execution engine. The engine extracts active packets from the FIFO execution engine queue. The TOS field of the active packet is in the form of the service name. At the active node, the service name of the active packet is looked up in the hash table and one of the following cases could occur.

Case 1: The requested service is not found in the hash table. This implies that the service has not been loaded. The EE attempts to load the service into the node.

Case 1a. If service loading is successful, the hash table is updated and the *process* routine of the service class is invoked on the active packet. Henceforth, all successive active packets requesting this service are directly processed.

Case 1b. In the current implementation, if the service loading is unsuccessful, the packet is discarded. In future implementations, we could make the node perform traditional forwarding on active packets that it cannot service. This implies building services that need not require processing at all intermediate active nodes.

Case 2: The requested service is found in the hash table. This implies that the service has been previously loaded and so the execution engine directly invokes the *process* routine of the service class returned by the hash table, on the active packet.

New services are uploaded to the active node through 'trusted operators'. A discussion of the security implications on designating these operators and implementing such a scheme is beyond the scope of this thesis. Figure 3.7 illustrates the functioning of the process module.

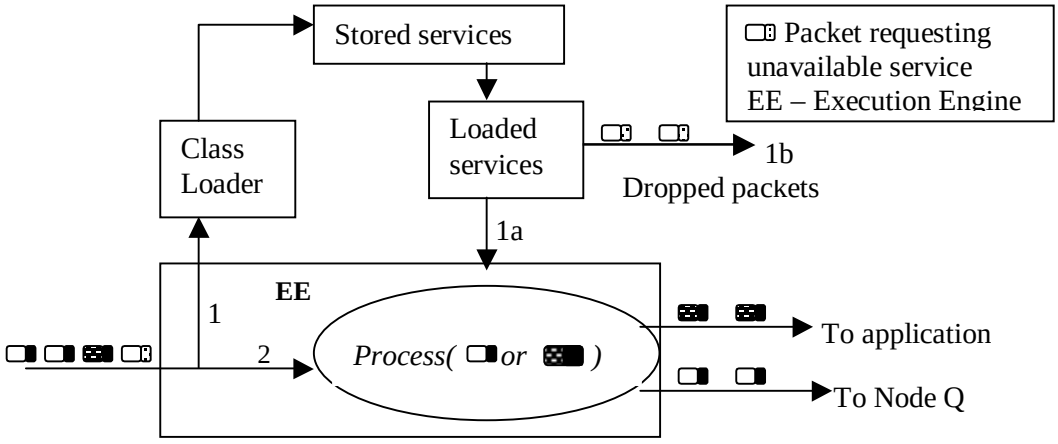

Figure 3.7 Process module

3.2.6 Transmission module

The node queue may receive packets from three sources. The first source is the front-end user interface (described in section 3.3) through which users may inject packets in the network. The second source is the receive module which may add passive packets that require forwarding. Lastly, the execution engine adds active packets that require forwarding to the node queue. The transmit module extracts packets from the FIFO node queue. It then looks up the routing table with the destination address and port number of the packet as the key to the table. The table returns[3] the virtual link on which the datagram must be sent. The node then converts the packet into a UDP datagram in byte array format and sends out the datagram on the returned link. To handle the special case of looping back (source and destination node fields are the same) of passive packets in the RANI node, a destination check on the passive packets is performed in the transmission module.

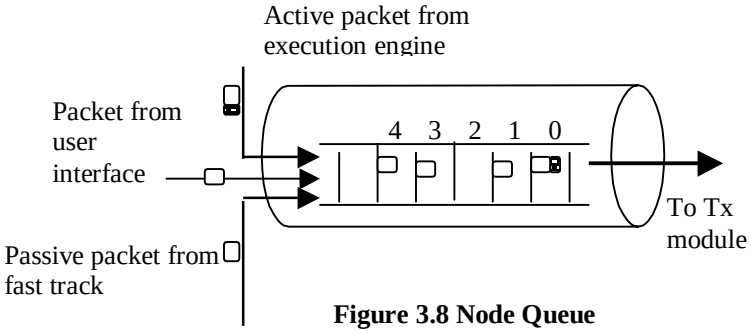

Figure 3.8 Node Queue

3.3 Node operation and configuration

We have provided a user-friendly GUI for configuring and operating the active node. Node operation includes injecting active or passive packets into the network, monitoring the node queues and testing virtual links for operation. Multiple packets can be injected with the help of a packet generator that simulates UDP or TCP-like sources. The user can select the number of packets, the average rate of injection of packets and burst size of the packets[4]. Node configuration involves creating (and destroying if necessary) virtual links, managing the routing table and

[3] If the routing table returns null, the destination is unreachable and the packet is discarded
[4] These parameters are described in Chapter 5

setting the queue parameters. All nodes in the active network are identified by unique IPv4 addresses. At run time of the active node, virtual links to other nodes are created through the user interface. Each link successfully created is automatically added to the routing table. The routing table is implemented as a hashtable containing the destination address and port number as the key and the virtual link object as the value. The table is automatically updated when new links are created or existing links are destroyed. We also provide access to manually configure the routing table for dynamically changing routes in the network. Shown below is an illustration of the user interface to our active node.

Active node interface

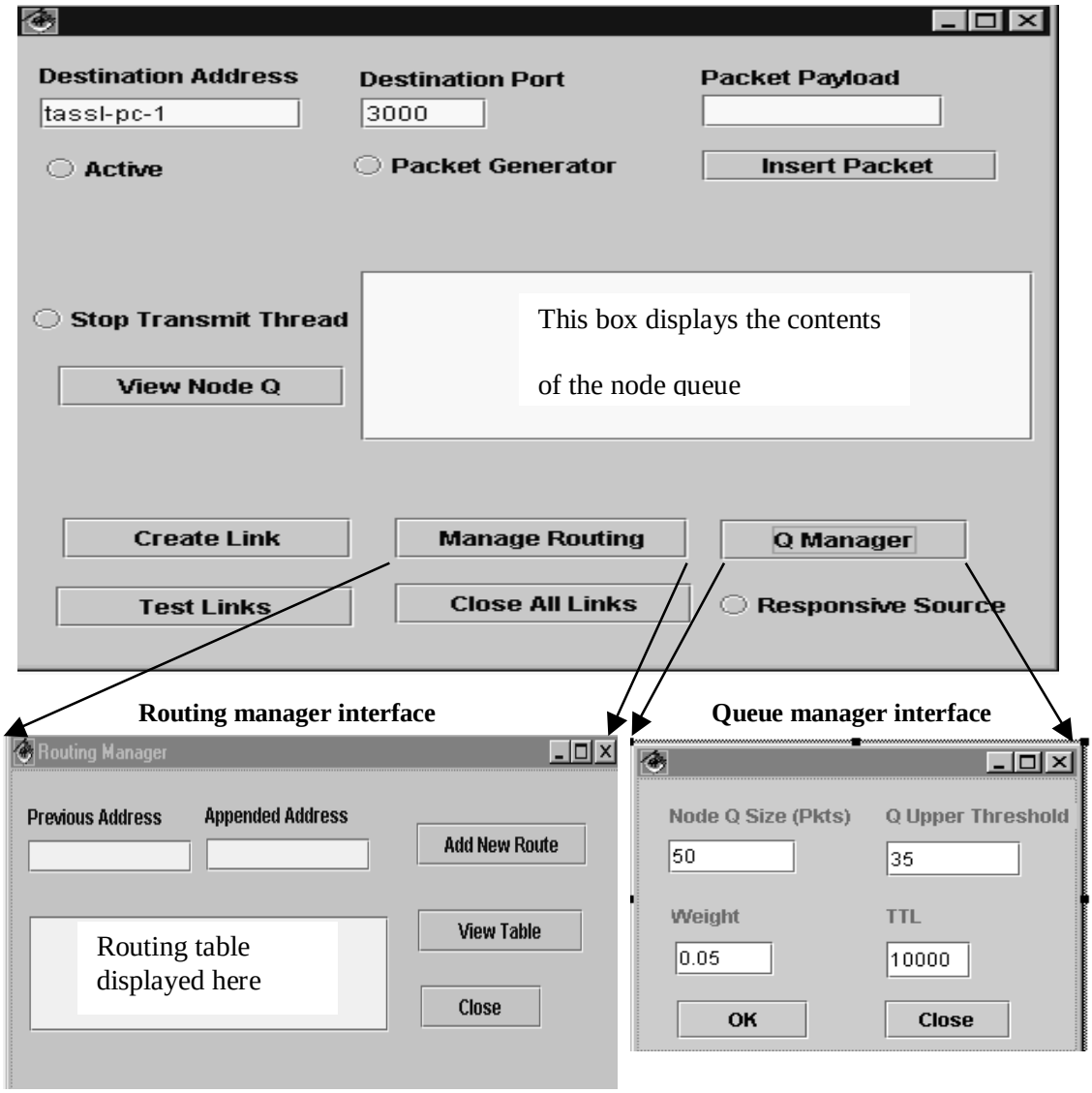

Figure 3.9 Graphical User Interface

To explain the construction of routing tables at our active node, consider the following network topology. Virtual links are labeled VLINK.

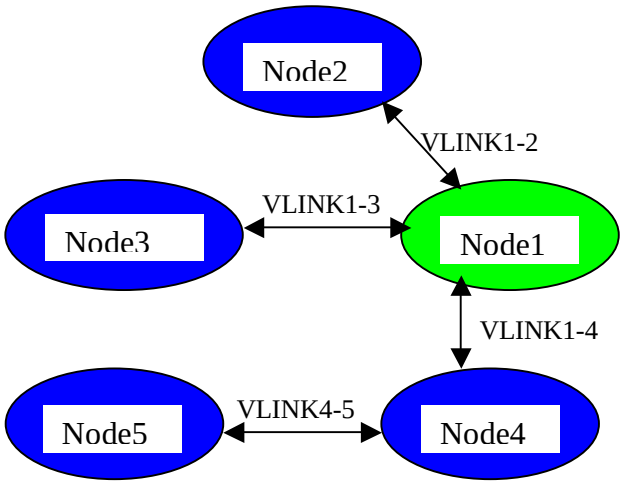

Figure 3.10 Sample network topology

In this example, the routing table constructed for node1 and node4 are shown below. We have assigned arbitrary IP addresses to the active nodes and used arbitrary port numbers for the virtual links.

NODE 1		NODE 4	
Key	Value	Key	Value
Node2 128.6.43.52:2000	VLINK1-2	Node1 128.6.43.20:6000	VLINK1-4
Node3 128.6.30.3:4000	VLINK1-3	Node2 128.6.43.52:2000	VLINK1-4
Node4 128.6.21.18:3000	VLINK1-4	Node3 128.6.30.3:4000	VLINK1-4
Node5 128.6.21.19:3000	VLINK1-4	Node5 128.6.21.19:3000	VLINK4-5

Table 3.1 Sample routing table

3.4 Summary of network features

- The active node provides an environment for communicating with applications, packet processing and network communications.
- The active node does not maintain state or flows unless programmed to do so for a specific purpose.

- The programming model of the network is based on a menu-based approach. End-users may request network processing through a service-ID field in the active packet. Trusted operators are allowed to load new services or enhance existing ones, thus minimizing security risks.
- At run time, the active node has a user-friendly graphical user interface through which it can be configured and operated. Also, the active nodes may be restarted and links may be dynamically changed to reflect a new network topology.
- In order to allow multiple packets to be processed simultaneously at the node, entities interacting with the packet such as queues, tables, threads, links and routines are synchronized.
- Separate tracks are maintained for active and passive packets to speed up traditional forwarding.
- The TTL field in the packet ensures an upper resource limit on the time that a packet may spend in the network.

3.5 Limitations of our architecture

Firstly, by processing packets within the active network the speed of packet transfer from end to end is reduced. Although end-applications may benefit from this additional network support even at reduced packet rates, it is important to maintain a high rate of packet transfer to prevent large packet queues from building up at the active nodes. Our active node is built at the application layer in the TCP/IP protocol stack. This makes its operation relatively slow. Secondly since the aim of this work was to examine active network technologies with respect to network utilities and congestion control we made simplifying assumptions such as reliable, bi-directional virtual links and static routing tables. These assumptions prevent real world scenarios from being simulated for other applications. Lastly, by allowing only trusted operators to load new services into the active we compromise on the dynamism in enabling new active services. However, these limitations do not undermine the contributions made in this thesis with respect to our objectives.

Chapter 4

RANI Applications

This chapter is divided into two parts. The first part describes the implementation and operation of RANI network utilities. The second part of this chapter addresses bandwidth greedy connections in the RANI network.

4.1 Host Reachability

In this section we first describe the implementation of the Ping network utility on traditional IP networks and then describe its implementation (APing) on the RANI testbed. Aping was the first active service that we developed as a sanity check for the RANI testbed.

4.1.1 Ping

The word “ping” stands for Packet InterNet Groper. The ping program is often used to test the reachability of another host on the Internet by sending it echo requests that it must respond to, if the host is operational [39]. The traditional ping program is one that sends an ICMP (Internet Control Message Protocol) echo request message to a host and waits for a reply. ICMP messages are encapsulated in IP datagrams and hence the operation of ICMP does not depend on the higher-level protocols such as TCP and UDP. Most TCP/IP implementations provide a ping program and it has proved to be a useful tool.

4.1.2 APing

The operation of APing along with its service routine is provided in this section. The APing active packet originates from a source node (S) that wishes to discover whether some other target node (T) in the network is alive. Intermediate nodes forward this active packet towards the target node. On receiving the active packet, the target node sends back an acknowledgement to the source. The source node (S) on receiving the acknowledgement displays a message saying that the queried host is alive. Assuming that the APing service is loaded at an active node, when an active packet requesting this service enters the execution engine of the node, the process method

of the APing class is invoked with the active packet as the formal parameter. A line by line description of the APing.process( ) method is given below.

```
process ( ActivePacket )
{
        if ( ! ActivePacket.destinationReached( ) ); // intermediate node reached
        {
                forward (ActivePacket); // packet forwarded to destination
        }
        else // Destination or Source node reached
        {
                if ( ! ActivePacket.getAck( ); // packet at Destination node
                {
                        ActivePacket.sendAck( ); // Create and return an acknowledgement
                }
                else // packet back at Source node
                {
                        printSuccess( ) ; // displays reachability message
                }
        }
}
```

4.2 Route Discovery

In this section we first describe the operation of the traceroute utility on traditional networks. Then we describe its design and implementation on the RANI testbed concluding with a comparison of the two implementations.

4.2.1 Traceroute

Traceroute allows users to discover the route of an IP datagram from a source node to another node. Traceroute uses the ICMP ‘time exceeded’ message and the TTL (Time To Live) field of the IP header. The utility requires end nodes to have a programming interface to the TTL field of an outgoing datagram. Availability of this programming interface to many networked nodes and simplicity of its operation make this utility popular in TCP/IP networks. Traceroute operates by

sending UDP datagrams to the destination node with the destination port number selected to be of a large value (>30000) making it highly improbable that an application at the destination is using that port [38]. The utility begins operation by sending a UDP datagram towards the destination with a TTL set to 1. The first router to receive the datagram, decrements the TTL to 0, subsequently discards it and then sends back an ICMP '*time exceeded*' message to the source. The source node thus identifies the first router in the path to the destination. Now, traceroute sends a UDP datagram with a TTL of 2, thus discovering the second router in the path to the destination node. This process continues till all routers upto the destination node is identified. When the destination receives a datagram with the TTL of 1, it does not discard it since no further forwarding is required. Instead, the node attempts to deliver the datagram to the 'unusually high' port number which is almost certain to be unused by any application. This results in an ICMP *'port unreachable'* message being sent back by the destination to the source node. The utility running at the source node distinguishes between the ICMP '*time exceeded*' and '*port unreachable*' messages to terminate route tracing.

A technical point overlooked above is that for each value of TTL, the utility sends three datagrams and prints the roundtrip times of the received ICMP messages. If no response is received within 5 seconds, the utility prints an asterisk and continues operation.

Note:

- The traceroute utility assumes that consecutive datagrams from the same source to the same destination follow the same route.
- Time complexity of operation of traceroute is O (n^2) where n is the number of hops between source and destination nodes.
- Resource complexity in terms of links traversed is O (n^2).
- The source node transmits successive IP datagrams towards the destination with incremental TTL field values till the destination node is reached.

4.2.2 Atraceroute

4.2.2.1 Objectives

- To accurately determine the forward path of an active packet from a source node to any other node in our active network.
- To discover the node-resident time of the active packet at each active node in transit. The processing delay and queuing delay constitute the node-resident time of the active packet and enable us to determine the overheads involved in active processing.

4.2.2.2 Operational details

Atraceroute operates by injecting a single active packet requesting the Atraceroute service. This is expressed in the packet's Type of Service field. When the first active node in transit receives the packet, it forwards the active packet it received *and* sends back a description of its IP address and the packet processing time in the form of an active packet to the source node. The source node on receiving the descriptive packet discovers the first node in transit. When the second node receives the 'originating' active packet, it similarly executes the Atraceroute service by forwarding the received packet to the destination and sending back a descriptive packet to the source node.

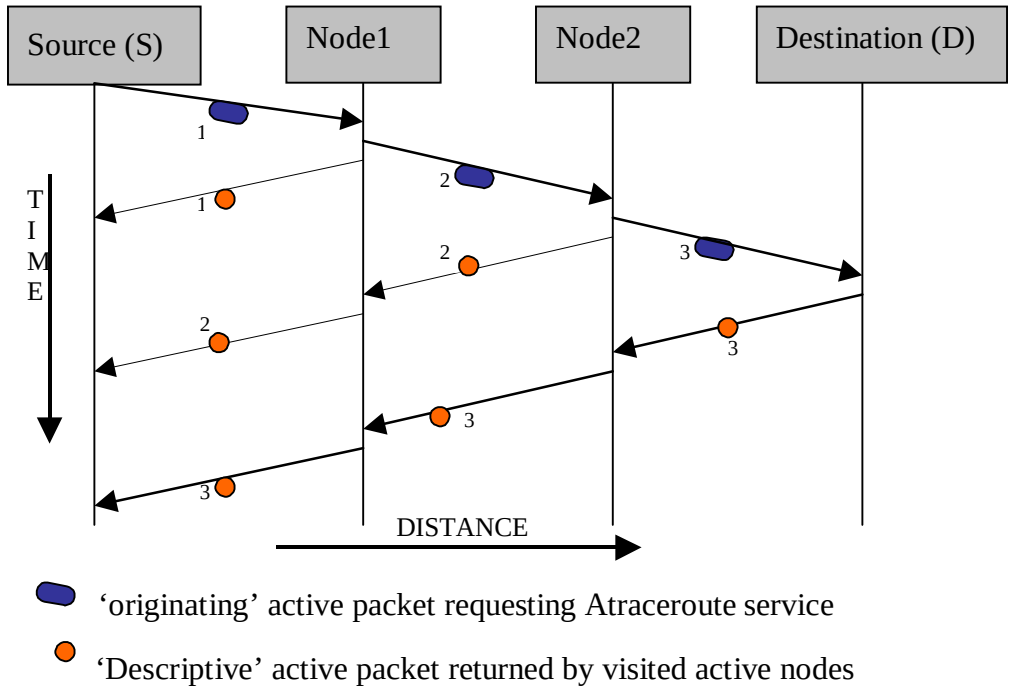

Figure 4.1 Operation of Atraceroute

This process continues till the destination node is reached. The destination node sends back a descriptive packet to the source and discards the originating active packet. Thus the source node discovers all active nodes in the path of the original active packet.

4.2.2.3 Taking a closer look at Atraceroute

- Figure 4.4 shows the forward and reverse paths of the active packet to be identical since our implementation assumes bi-directional virtual links. However this assumption is not necessary for successful operation of Atraceroute.
- The destination for the originating packet is the D node where as the destination for descriptive packets is the S node.
- The originating and descriptive packets both request Atraceroute servicing. But, the originating packet has its ack field set to false where as the descriptive packet has its ack field set to true. Active nodes in transit use the ack field to distinguish between originating and descriptive packets. A node receiving an originating packet creates a descriptive packet and sends it back to the source of the originating packet. A node receiving a descriptive packet simply forwards it to the destination.
- An interesting scenario would be to tackle re-ordering of packets at the source node in the eventuality that descriptive packets overtake each other on the return path. A possible solution could be to force the originating packet to carry state information regarding the node that it visits in the forward path. So when the originating packet is injected into the network it starts off with its state set to one. When Node1 receives this packet it echoes a descriptive packet carrying this state (one) and forwards the originating packet with the state modified to two. Now Node2 receives the originating packet with state two. Hence it echoes back a descriptive packet with this state, increments the state in the originating packet and forwards it. This process ensures that returning descriptive packets carry the corresponding number of the node visited by the originating packet making it possible to re-order the descriptive

packets at the source node. In Figure 4.4, the numbers mentioned on the packets represent the state information they could carry. However, in our implementation the bi-directional virtual link assumption prevents packets from being re-ordered in the network.

4.2.2.4 Implementation of Atraceroute

The process routine of the Atraceroute service is shown with appropriate comments below.

```
process ( ActivePacket )
{
        if ( ! ActivePacket.destinationReached( ) ); // intermediate node reached
        {
                if ( ! ActivePacket.getAck( ) ) // implies 'originating' packet received
                {
                        forward (ActivePacket); // 'originating' packet forwarded
                        sendDescriptivePkt( ); // 'descriptive' packet created and returned
                }
                else //  descriptive packet received from some node upstream
                {
                        forward ( ActivePacket ); // the packet is simply forwarded
                }
        }
        else // Destination ( Source or Destination ) node reached
        {
                if ( ! ActivePacket.getAck( ) ) // 'originating' packet at Destination node D
                {
                        sendDescriptivePkt( );
                }
                else // 'descriptive' packet at Source node S
                {
                        printPacketPayload( ); // Prints out contents of descriptive packet
                }
        }
}
```

4.2.2.5 Features

- The Atraceroute utility injects only one active packet into the network. Hence, we do not assume static routes from the source to the destination.
- Time complexity of operation of Atraceroute is O(n) and the link utilization is $O(n^2)$, where n is the number of hops between the source and destination nodes. Details are presented in Appendix A.
- The node resident time of the originating packet at each active node in transit is determined.

4.3 Network congestion and unresponsive connections

The past few decades have seen the merging of computers and communications leading to the development of computer networks. Rapid progress in technology coupled with the immense popularity of the Internet has seen an exponential growth in networked systems over the past few years. Formally, a computer network means an interconnected collection of autonomous computers [1]. The principle aim of a networked system has been information gathering, processing and distribution. Ideally, we would like to design and organize the network such that *all information* should be delivered *reliably* to *any networked location* within an *acceptable time frame*. Users of this ideal network would then derive maximum utility. However, the real world is far from ideal, leading to the development of networks that fail to satisfy one or more of the above criteria. A prominent cause that widens the gap between the ideal and real world scenarios is *network congestion*. Despite research efforts in industry and academia to eliminate network congestion, the problem continues to persist.

In [5] Yang and Reddy have broadly classified a range of congestion control algorithms into *open loop* and *closed loop* control mechanisms based on control theory. In the open loop algorithms, the transmitting sources carefully regulate the effective rate of transmission to prevent congestion from developing in the network. Such mechanisms cannot be relied upon completely due to the

dynamic nature of network traffic and network parameters. In the closed loop control mechanisms, it is the network that provides feedback to the transmitting sources either when it is congested or when congestion is building up. The transmitting sources then reduce their effective transmission rates in order to prevent clogging up the network. Both these mechanisms rely on the transmitting sources to exercise control. A growing number of applications require a constant rate of transmission (they cannot function without a minimum application-specific bandwidth requirement) while some others tend to 'grab' as much network bandwidth as available ignoring congestion notification. These applications fail to implement a transport mechanism that is responsive to the congestion status fed back to them from the network. Going by the nature of such applications we refer to them as 'unresponsive connections'. Formally, *an unresponsive connection is one that ignores or underplays feedback information regarding congestion status of the network.* Examples of such applications include streaming multi-media services, Voice transmissions and web radio broadcasts.

Internet traffic measurements taken in mid-April 1998 on OC-3 links within nodes on the iMCI backbone data have revealed "Web traffic constitutes 75% of the bytes, 70% of the packets and 70% of the flows when client and server traffic are considered together" [24]. Let us consider one constituent of Web traffic - streaming media applications. A recent article in the New York Times [34] claims, "*In 10 years, movies and commercial television might very well be carried over Internet channels. This increasing demand will add vast amounts of streaming traffic to the Internet and could lead to what Van Jacobson* (chief scientist for Cisco Systems Inc.) *calls "congestion collapse" – the Internet equivalent of gridlock*". The article continues to describe the bandwidth greedy nature of such applications. "*By its very nature, streaming media has to flow continuously to the user's computer, so it cannot follow the same traffic rules as conventional data. But even so, it is possible for packets of streaming data to interact civilly with other traffic on the Internet. The reason they do not, Jacobson said, is that streaming media providers have no incentive to comply with traffic rules.*"

4.4 Understanding a congested network

Although a large number of definitions for network congestion exist in computer literature, we consider the following to be precise.

- **A network is said to be congested from the perspective of user *i* if the utility of *i* decreases due to an increase in network load** (where utility refers to a users preference for a set of resources) [36]. In this definition congestion is classified as an end-user perception of the state of the network. If a specific user's demands on the network are not affected, even under highly loaded conditions, for him the network is still uncongested though other users whose utility may have been adversely affected will perceive the network to be congested.
- **If, for any interval of time, the total sum of demands on a resource is more than its available capacity, the resource is said to be congested for that interval** [26]. This definition uses a demand-supply relation to identify congested periods in the network. The demand consists of delivering information from end-to-end and satisfying user constraints such as allowable delays and reliability. The supply includes (but is not limited by) network resources such as buffer space, link bandwidth and processor speed. Only if all demands are met, the network is uncongested. Jain also explains with examples how congestion is in fact *worsened* by an ad-hoc increase in these network resources. Rather than considering congestion to be a supply related issue, we need to control it by a sound design strategy.

4.5 Background

4.5.1 Introduction

In this section we discuss two relevant schemes for congestion avoidance; Random Early Detection [27] and Explicit Congestion Notification [28]. RED gateways signal congestion by marking or dropping packets. ECN is a specific implementation of RED in which packets are marked to minimize packet loss during congestion at the gateway. RED has been proven to be ineffective in controlling bandwidth greedy connections as explained in Section 4.5.2. We aim to extend RED in order to control greedy connections using the RANI active network testbed.

4.5.2 RED (Random Early Detection) gateways

RED gateways have a packet queue that is closely monitored to detect the build up of congestion. Based on queue occupancy, the average queue length (avg) is computed using a low pass filter with an exponentially weighted moving average. The gateway notifies connections of congestion either by dropping or marking packets arriving at the gateway. If a packet arrives to a full queue it is discarded. The gateway has two pre-set thresholds called min_{th} (minimum threshold) and max_{th} (maximum threshold). With every arriving packet, the avg is computed and compared to these two thresholds. If avg is less than min_{th}, arriving packets are not dropped or marked. If the computed avg exceeds max_{th}, all arriving packets are marked or dropped. If the computed avg lies between min_{th} and max_{th}, the gateway notifies a connection of congestion with a probability that is roughly proportional to that connections share of the bandwidth through the gateway. The average packet queue size (avg) is computed as follows:

$$avg = (1-w)*avg + w*q$$

where

w < 1 is a queue weight that determines the degree of burstiness permissible by the gateway

q is the number of packets in the queue

The value of avg is computed with every packet arrival at the gateway. However, when a packet arrives to an empty queue (q = 0), avg is calculated differently. The gateway first calculates the idle time for the packet queue as the difference between the time at which the packet arrived and the time at which the queue length became zero. The average packet queue (avg) is then computed as if the gateway had transmitted m packets during the idle time. The factor m is linearly dependent on the time for which the queue was idle. Thus for an empty queue,

$$m = f(time - q(time))$$

$$avg = ((1-w)**m)*avg$$

where

q (time) is the time at which q became zero

time is the time at which a packet arrives to the empty queue

time – q(time) is the idle time of the packet queue

f ( ) is a linear function representing the rate at which the packet queue is drained

A detailed explanation of the RED algorithm can be found in [27].

The advantage of RED gateways is that they help in keeping the average queue size low, allow occasional packet bursts and prevent global synchronization of packet sources due to the randomness of the RED algorithm in marking or dropping packets at a congested node. However, it has been proven through simulations that an unresponsive bandwidth greedy connection gets a larger than fair share of the bandwidth at a RED gateway when competing with responsive connections [2]. But the congestion avoidance schemes suggested in [2] require multiple queues to be maintained at the intermediate nodes of the network. We propose a mechanism using the RANI network to maintain a single FIFO queue at the intermediate active nodes.

4.5.3 ECN (Explicit Congestion Notification) capable gateways

Explicit congestion notification [28] is a mechanism that notifies transmitting sources of incipient congestion by setting a bit in the IP header of the packet (called packet marking). When the marked' packet reaches the destination, congestion notification is echoed back to the sender via the acknowledgement packet. The sender is then expected to cut back the packet transmission rate. However, the connections need to be ECN capable; the end hosts must be capable of responding to marked packets for the scheme to work.

4.6 Aims of our congestion control strategy

- The algorithm must be simple and easily deployable in the RANI active network testbed. Congestion leads to performance degradation of a network. Deploying a complex algorithm would amount to consuming network resources at a time when resources are scarce.

- The designed algorithm must be efficient and effective. An efficient algorithm would have minimal overheads. The effectiveness of the algorithm must be justified through experimentation.
- The algorithm must accurately detect bandwidth greedy connections at a congested node. In section 4.3 we highlighted the growing popularity of unresponsive connections. However, it is important to note that unresponsive connections are not necessarily bandwidth greedy. If that were the case our algorithm would restrict all UDP connections in the active network. Our aim is to limit the degradation in network performance caused by transport mechanisms that tend to increase or maintain their effect rate of transmission of packets, despite being asked to cut back during periods of congestion.
- The algorithm should provide a negative incentive to greedy connections in order to limit their popularity.
- The algorithm must scale well. It should be capable of handling multiple greedy connections through a congested node.

4.7 High level design of algorithm

Our congestion control strategy is optimized for the reservationless packet switched RANI active network described in Chapter 3 and could be implemented in other active network architectures as described in Chapter 2. The high level design is illustrated as a flow chart in Figure 4.1. We have used the words flow and connection interchangeably and have described the characterization of a flow in the implementation details.

- **Monitoring the system to detect congestion**: When the demand on the network exhausts its resources, the network nodes are the first to be affected. Specifically, when a node gets congested the packet queue gets heavily occupied eventually forcing the node to drop packets that overflow the queue. Hence, packet queues at the intermediate nodes in a network are the ideal location for detecting the build up of congestion.

- **Distributing the congestion-related information to places that can control deterioration of system performance**: We divide the set of connections through a congested node into two distinct categories viz. non-greedy and greedy connections. Non-greedy sources either respond to congestion notification or do not make a heavy demand on network bandwidth during congested periods. In the case of non-greedy sources the control loop is stretched from the congested node to the packet source. We rely on RED mechanisms and the packet source to reduce the rate at which packets enter the congested node. Bandwidth greedy sources underplay or ignore the fed back congestion related information in the form of dropped or marked packets. Controlling such sources is the focus of our algorithm. Stretching the control loop to the packet source is ineffective and hence congestion caused by greedy sources is controlled at the congested node itself and not by relying on the greedy sources to cut back their effective rate of packet transmission.
- **Correcting system operation**: Demand on a network node is gauged by the effective rate of arrival of packets at the node. In order to eliminate congestion at a node, the effective rate of packet flow through the node needs to be reduced. During severe congestion, the packet arrival rate from greedy connections is controlled by a mobile filtering mechanism. In this mechanism a packet filter is installed at the congested node for the identified greedy connection. The filter is then progressively migrated towards the source of the greedy connection using active messages. In doing so, the packet drops are made early and causes lesser wastage of network resources. Filtering packets belonging to a flow is a relatively harsh mechanism of controlling congestion but is deemed necessary, taking into account the damage that can be done to network resources by the greedy connection. Keeping in mind that multiple flows could be identified as bandwidth greedy, we pick out the greediest flow and dynamically filter packets belonging to it. However, if congestion is not controlled despite filtering the greediest flow, the algorithm continues to successively pick out flows in order of their greediness.

In summary, our algorithm must first detect when a node's packet queue is about to overflow due to increased demand. It must then correctly identify greedy connections (if any) that may be responsible for this extreme condition. Subsequently, by using the processing capabilities of the active network nodes in the path of the greedy connection(s), the algorithm must effectively control the rate at which packets from the greedy connection(s) enter the congested node.

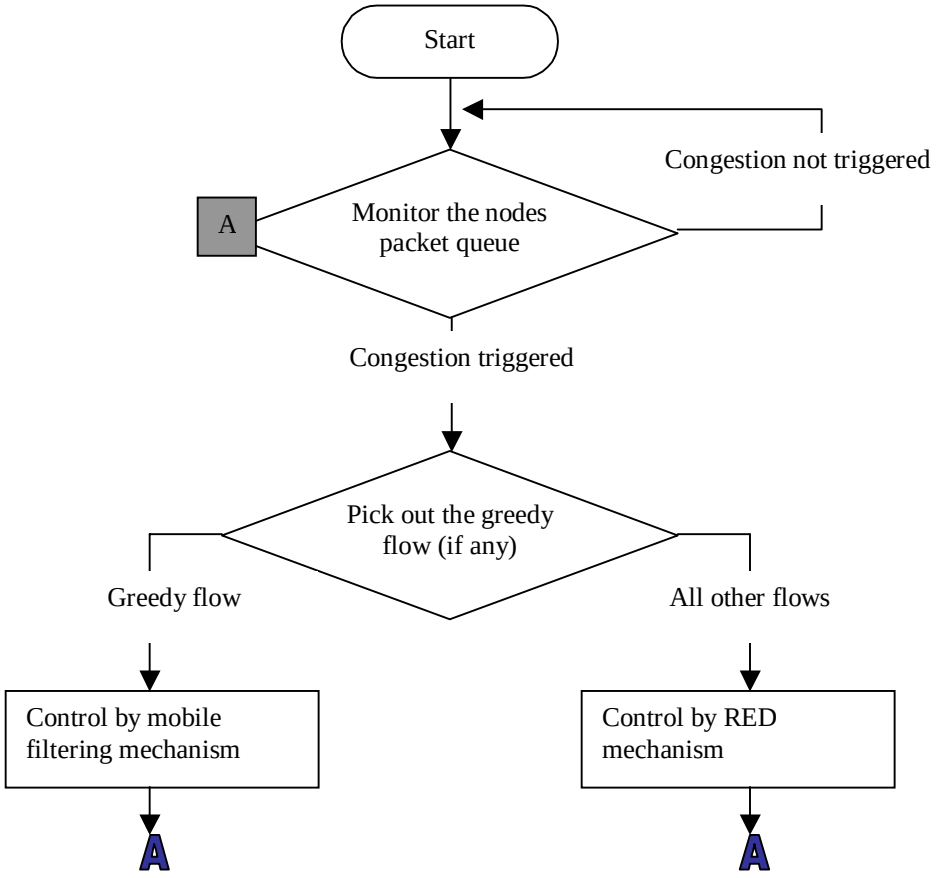

Figure 4.2 Flowchart for high level design of our congestion control algorithm

4.8 Implementation

We have labeled our implementation of the algorithm described in section 4.5 as the LGC (limiting greedy connections) algorithm.

4.8.1 Monitoring the active node's packet queue to isolate a greedy connection

In [9], Dong et. al have demonstrated through simulations that in RED gateways the bandwidth consumed by greedy connections is greater than its fair share. In fact, bandwidth consumption is directly related to the queue occupancy of the connection. A connection with a large share of

bandwidth consumption on a link has a correspondingly larger share of packet queue occupancy at the node. Thus, we use queue occupancy metrics to detect a greedy connection.

In RED gateways when avg exceeds max_{th}, all packets arriving at the node are marked or dropped. In this state, the nodes packet queue is close to overflow and we label the node to be in a 'severely congested' state. We have observed that maximum disparity between queue occupancy for non-greedy and greedy connections occurs at this time. To ensure accuracy in identification of greedy connections, our algorithm is triggered in the severely congested state of the node. For simplicity we identify a connection by a source IP address, source port tuple although it would be more accurate to identify connections by a source IP address, source port number, destination IP address, destination port number, IP protocol tuple.

To identify the greedy connection at a severely congested node, first we need to determine the fair share (f) of a packet queue. Consider an active node having a total packet queue occupancy of 75 packets with 5 connections competing for a share of the bandwidth. The fair share in terms of packet queue occupancy would be given by

f = Total queue occupancy(p) / number of connections represented in the queue(n) [a]

i.e. f = 75/5 = 15 packets

Ideally, to ensure a fair distribution of bandwidth, each connection should not have more than 15 packets buffered at the node. But a responsive connection may have more than its fair share of packets buffered at the node due to several reasons. Some of the prominent reasons cited in [29] are the bursty nature of Internet traffic, a possibility of high delay-bandwidth links on the receive port of the node and connections being in different phases of operation. We provision for these discrepancies by a factor 'k' > 1. The value for k is selected to be $\log_e(3n)$. The factor k decides the degree of permissible disparity between greedy and non-greedy sources. Selecting a small value of k may cause the algorithm to wrongly classify a responsive source as greedy, where as selecting k to be too large will make it nearly impossible for the algorithm to detect a greedy connection. A similar value is chosen in [29] for identifying flows using disproportionate

bandwidth. However that scheme also relies on the characterization of a conformant TCP source based on an assumed value of round trip time for the connection. Our approach to detect an unresponsive connection is purely based on the queue occupancy of the connections when a node is severely congested.

Assuming that the separation between min_{th} and max_{th} is large, avg is unlikely to increase from min_{th} to max_{th} before providing ample time for the responsive connections to back off. In this scenario, when average queue size exceeds the maximum threshold, and a large disparity occurs between queue occupancies of competing connections it is safe to assume that the connection with an exceptionally large number of packets buffered at the severely congested node is bandwidth greedy. Continuing with our example, $k = \log_e(15)$ or $k = 2.708$

We calculate the responsive share (r) of the packet queue occupancy as

$$r = \lceil k*f \rceil \qquad [b]$$

$$or\ r = 2.708 * 15 = \lceil 40.62 \rceil$$

So in this example, a connection that has at most 41 packets in the queue (i.e.54.66% of queue occupancy) during its severely congested state is assumed to be responsive. All connections having more than a responsive share of the packet queue are assumed to be unresponsive. Amongst the unresponsive connections identified, the one having the maximum number of packets buffered at the severely congested node is singled out as the 'greedy' connection. Combining (a) and (b) we have,

$$r = \lceil (\log_e(3n)*p)/n \rceil$$

$$i.e.\ r = \lceil p*(\log_e(3n))/n \rceil$$

The permissible queue occupancy expressed as a percentage is then given as:

$$qo = 100*r/p = 100* \log_e(3n)/n \qquad [c]$$

Figure 4.2 shows the permissible queue occupancy expressed as a percentage (qo) plotted against the number of connections (n) represented in the queue. The slope of the graph is steep for

smaller values of n and becomes a gradual decline as n increases. This implies that a larger variation in queue occupancy is permitted when fewer connections cause severe congestion at a node. One anomaly that appears is that for the special case of n=1, a connection will not be classified as greedy even if it exhausts the entire packet buffer at the node. This is in fact necessary, so that a single connection will never be filtered, since there is no competing connection.

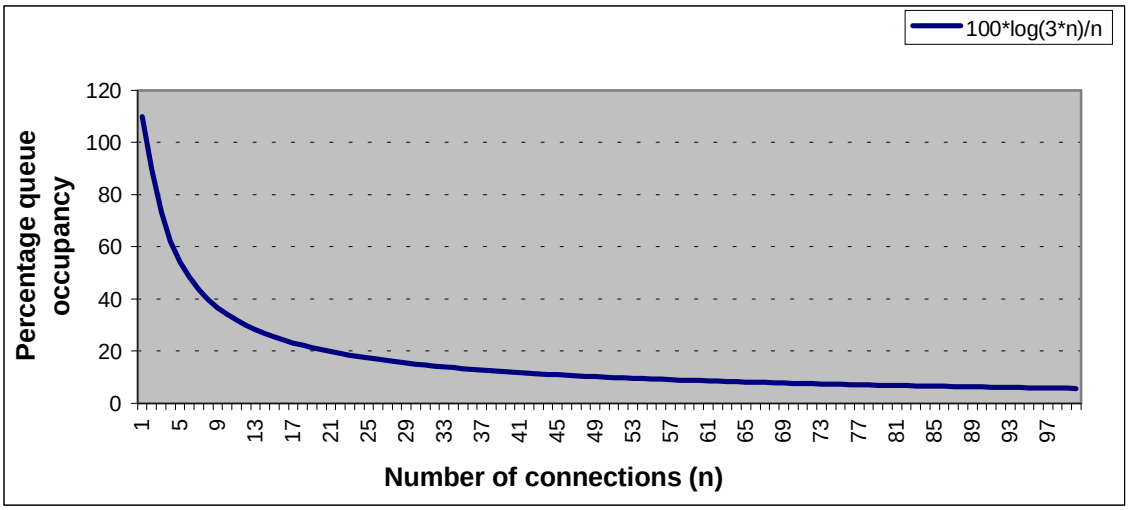

Figure 4.3 Percentage of permissible queue occupancy v/s number of connections

4.8.2 Controlling the greedy connection

When the node is severely congested, reducing buffer occupancy is of utmost importance or the buffer will overflow causing all arriving packets to be dropped and the RED gateway will reduce to a drop-tail gateway. The disadvantages of drop-tail gateways are explained in [4, 27]. To prevent the node to degrade into a drop-tail node, it becomes imperative to prevent the buffer from overflowing. We feel that the only effective way to control the inflow of packets from a greedy connection is by actively filtering packets belonging to the connection. The packet filtering must continue until such a time that the queue occupancy of the packet buffer at the severely congested node is reduced to acceptable levels. Once this happens the responsive connections may compete for a fair share of the bandwidth that they were previously denied.

Also, the packet filtering can take place anywhere along the path of the connection from the source to the congested node.

We control greedy connections by a process of mobile filtering. A packet filter for the greedy connection is installed at the congested node. This filter migrates towards the source of the greedy connection and stops at the first hop node of the connection. At the first hop node, the packet filter is installed for a pre-programmed interval of time. In our implementation migration of the filter is possible due to the assumption of bi-directional virtual links. In future implementations, the PrevHop field of the packet may be used to move the packet filter towards the source of the greedy connection.

4.8.3 Operation of the mobile filter – The active filter service

The process of mobile filtering begins with the congested node extracting a packet belonging to the greedy connection from its packet buffer. This packet reveals the source of the greedy connection. A *greedy connection identifier* (GCI) consisting of the source IP address and port number is formed. Next, the virtual link object connecting the congested node to the greedy source is obtained from the routing table using the GCI. The node uses the GCI to create a packet filter on the receive thread of the virtual link. The packet filter drops packets originating from the identified greedy connection. The virtual link object reveals the active node to which it connects. The IP address and port number of this active node is called the *previous hop identifier* (PHI). The node then creates an active packet destined for the previous hop requesting the ActiveFilter service.

Figure 4.3 shows a network topology to illustrate the operation of LGC. Consider a greedy connection G identified (by the procedure described in section 4.8.1) at the severely congested node N4. This connection G competes with four other responsive sources (R) for link V9. N4 extracts a packet belonging to G from its packet queue and forms the GCI. Using the GCI and by looking up its routing table, node N4 learns that the packet was received over link V6. N4 creates a packet filter on the receive thread of link V6 to drop packets belonging to G. Link V6 reveals

that it is connected to node N3. In effect, N3 is the previous hop node for the identified greedy connection G. Now N4 sends an active packet to N3 with its payload carrying the GCI, requesting the active filter service. N3 on receiving the active filter message similarly installs a packet filter for the mentioned GCI and propagates the active filter message to the next hop closer to G's source that is to node N2.

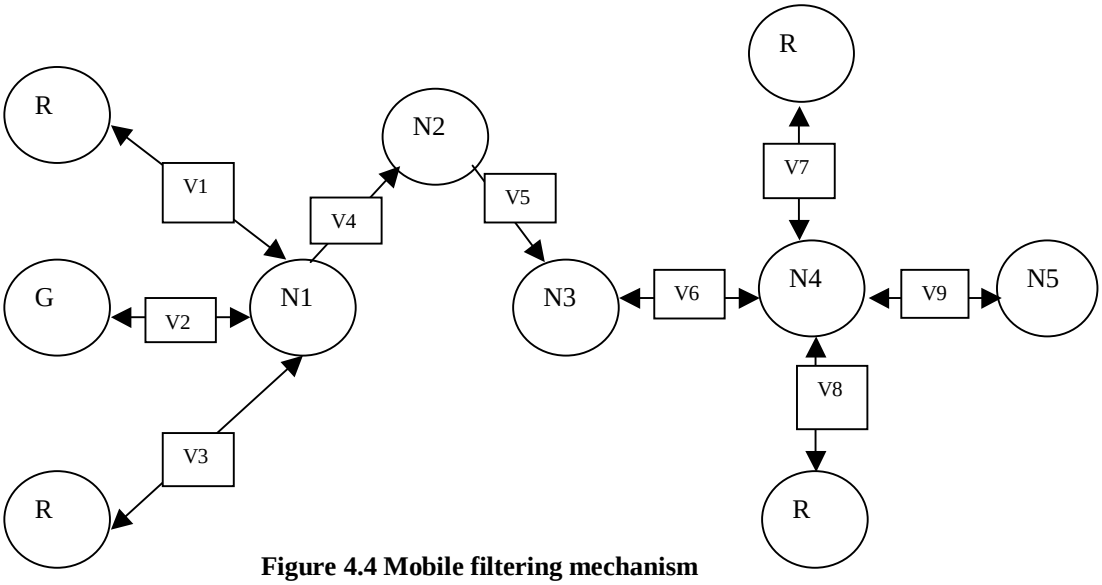

Figure 4.4 Mobile filtering mechanism

This process continues till the first hop node N1 for the greedy connection is reached. A minor technicality overlooked in the example above was the assumption that a node can automatically learn if it is the first hop node and stop propagating the mobile filter. This is because prior to creating the active filter message each active node performs a previous hop check. The check consists of a comparison of the GCI and the PHI fields. If they match it means that the filter has reached the first hop node for the connection G. The packet filter is then installed for a longer duration of time and the node does not propagate the active filter message any further. Sending the active filter message to the source of a greedy connection would be futile for reasons explained in section 4.3. Continuing with the above example when the active filter message reaches node N1, the Previous Hop Identifier and the Greedy Connection Identifier are both G. Thus no further active filter messages are sent in the network.

Once a greedy connection is identified and filtered at the congested node the packet queue occupancy is expected to drop. However, due to the low pass filtering mechanism used in the calculation of avg, its value might continue to be greater than max_{th} even after the queue occupancy has decreased. This will again trigger the LGC algorithm. To ensure that LGC is not triggered multiple times in a short interval of time, a minimum idle period is chosen between two consecutive triggers of LGC.

4.8.4 The LGC algorithm

Variables used

- avg – Calculated average queue size
- maxth – Upper threshold for node queue
- Suspend_LGC – A Boolean variable used to ensure a minimum idle time between consecutive triggers of LGC
- ITime – The time for which the packet filter for the greedy connection is installed at an intermediate node
- FHTime – The time for which the packet filter for the greedy connection is installed at the First Hop Node.
- Tx – The minimum idle time between successive triggers of the LGC algorithm

Initialization

avg = 0, Suspend_LGC = false

maxth, lTime, FHTime and Tx are pre-set and configurable.

The average (avg) is calculated when a packet is added to the packet queue. The LGC algorithm is shown below:

If (avg ≥ maxth && (!Suspend_ESC)) {
Set Suspend_ESC to true for Tx
Find out responsive share of packets for a connection (r)
Determine the greedy connection and corresponding GCI
Determine virtual link on which its packets arrive and PHI

```
        If (GCI != PHI){
                Install filter for GCI for ITime on virtual link
                Send filter message to previous hop node
        }
        Else
                Install filter for FHTime for GCI
}
```

The process routine of the active filter message is shown below:

```
Process (active_filter message) {
        Extract GCI from payload of active packet received
        Determine the virtual link on which its packets arrive and PHI
        If (GCI != PHI){
                Install packet filter for GCI for ITime on virtual link
                Send filter message to previous hop node
        }
        Else
                Install filter for FHTime for GCI
```

4.8.5 Importance of timing parameters in LGC

ITime: Referring to Figure 4.3 lets examine the sequence of events during the migration of the filter from node N4 to its previous hop node N3.

a) Packet filter installed at N4 at time t0.

b) Packet filtering begins at N4 at time t1

c) Active message sent to N3 at time t2.

d) Active message reaches N3 at time t3.

e) Filter installed at N3 at time t4.

f) Filtering begins at N3 at time t5.

Once the filter is installed at N3 for the greedy connection it may be discarded at N4. The time it takes for the filtering operation to migrate from N4 to N3 is T = t5 – t0. So, after time T the filter may be discarded at N4. However, the time it takes for the message to be propagated from a node

to the previous hop node (t3 –t2) is dependent on physical characteristics of the network. Thus we set ITime to about 5 seconds for our implementation assuming that t5-t0 < 5 seconds. Further, by selecting a slightly large value for ITime we can be sure that the packet filter for the greedy connection will be installed at node N4 until such a time that all packets belonging to that connection are drained from node N3.

FHTime: When the filter reaches the first hop node, it stops migrating and is then installed for FHTime seconds. If we keep FHTime too small the unresponsive connection will not be filtered long enough and could congest the network again. If we keep it too large the connection may close but the packet filter will continue to exist adding unnecessary overheads at the gateway at which it is installed. We set FHTime to about 100 seconds in the RANI testbed.

Chapter 5

Experimental Evaluation

5.1 Evaluating the LGC algorithm

In this chapter we evaluate the utility of the mobile filtering mechanism and the LGC algorithm by experimentation on the RANI testbed in a configurable environment, followed by an analysis of the results obtained. Since the LGC algorithm is triggered only during severe congestion, this state of the node becomes the starting point for our experimental evaluation. In all our experiments we force a node into severe congestion and observe the relevant values of the node parameters to deduce the events occurring at the node. The aim of the LGC algorithm is to control greedy connections. Also, in order to reduce complexity in the implementation we do not implement the RED algorithm in its entirety.

5.1.1 Experimental Environment

The machines used in the experiment had an Intel Pentium II 300 MHz processor. These machines were interconnected via a 10BaseT Ethernet LAN at the data link layer. The RANI network was built on the Windows NT operating system substrate.

5.1.2 Source simulation

To bring out the effectiveness of the LGC algorithm we simulate responsive and unresponsive connections. The sources are simulated with the help of a packet generator that can be selected to behave as a responsive or a greedy source.

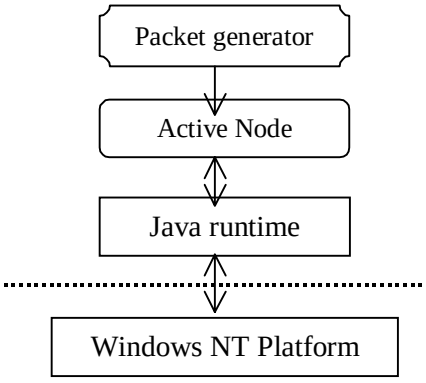

Figure 5.1 The active node at runtime

The transport mechanism for a responsive connection is simulated as a rough approximation of a TCP source. A detailed explanation of the implementation of TCP can be found in [22]. In this section we briefly describe our implementation of the relevant parameters of the TCP window. Our TCP-like source contains common parameters as implemented in TCP such as the slow start threshold (ssth) and congestion window (cwnd). The slow start threshold parameter for the responsive source is set through the user interface. The packet generator begins in a slow start phase in which the congestion window (initially set to one) is doubled every round trip time (similar to TCP's exponential increase in cwnd) until it equals the threshold. Now the generator enters the congestion avoidance phase in which the congestion window is incremented by one packet every round trip time (similar to TCP's linear increase in cwnd). The transport mechanism for an unresponsive connection is simulated by a constant packet-rate source. The end user can configure the total number of packets, the number of bursts and the inter-burst spacing in milliseconds through the user-interface. For example the end user may select the total number of packets as 200, the number of bursts as 8 and the inter-burst spacing as 300msec. Correspondingly, the packet generator will inject 25 back to back packets, pause for 300 milliseconds, inject the next 25 packets back to back, then pause again for 300 milliseconds, and so on until all 200 packets are sent.

5.1.2 Network Topology and active node parameters

The active network topology comprises of the number of source nodes, interconnecting nodes, sink nodes and the virtual links interconnecting these nodes. The active node parameters consist of the time-to-live field set in the packets and the node queue parameters defined by the size of the buffer, the weight (w) used for calculating the average queue size and the maximum threshold (max_{th}) of the nodes packet queue (Node Q).

For each experiment we select a topology that tests a particular aim of the LGC algorithm and select node parameters such that at least one of the intermediate nodes gets severely congested in

order to trigger the LGC algorithm. To prevent packet loss we select a large buffer size at the intermediate node targeted for severe congestion.

5.1.3 Presenting results

Results are presented in the form of graphs, tables and statements for the following:

- The throughput observed for each of the sources is expressed as a percentage of packets successfully reaching the destination node
- Installation and mobility of active filters (if any) for the identified greedy connections
- Actual queue size measurements and average queue size measurements for a given set of active node parameters.

5.1.4 Experiment 1 – Basic operation

In this experiment we test the ability of the LGC algorithm to correctly identify and filter a greedy connection. The test network consisting of six responsive sources, one greedy source, one interconnecting node and a sink node is shown in Figure 5.2.

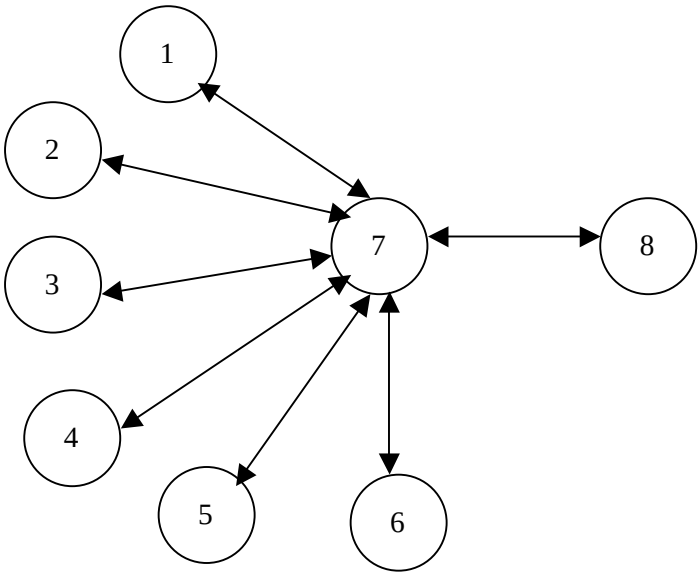

Figure 5.2 Network topology for Experiment 1

Node 1 is the greedy source and nodes 2,3,4,5,6 and 7 are responsive sources. Node 7 behaves as a responsive source and is targeted for severe congestion. Node 8 is the common sink for all the sources. Virtual links are shown as double-ended arrows. Node 7 is forced into a severely

congested state by having all the sources transmit packets at approximately the same time. To prevent packet drops due to expiration of the TTL field, all packets injected into the network have an initial TTL of 10 seconds. The queue parameters for node 7 are set with queue weight = 0.02, max_{th} (Upper threshold) = 25 and buffer size = 50. The responsive sources inject 50 packets each with an initial TCP slow-start threshold set to 16. The greedy source injects 200 packets in 5 bursts with an inter-burst duration of 1 second.

In Figure 5.3, the x-axis shows the packets arriving at node 7 and the y-axis shows the queue size measured in packets. The solid line (y = 25) represents the configured value of max_{th} at the node. Notice that the low pass filtering mechanism of RED causes the average queue size to change slowly in comparison to the actual queue size. For brevity, the first few packet arrivals have been omitted in Figure 5.3. Initially as the responsive sources open up their windows, the actual queue size remains low (<10). Once the competing sources have sufficiently large windows, the actual queue size increases rapidly. When the average queue size crosses max_{th} viz. 25 in this case, the LGC algorithm is triggered.

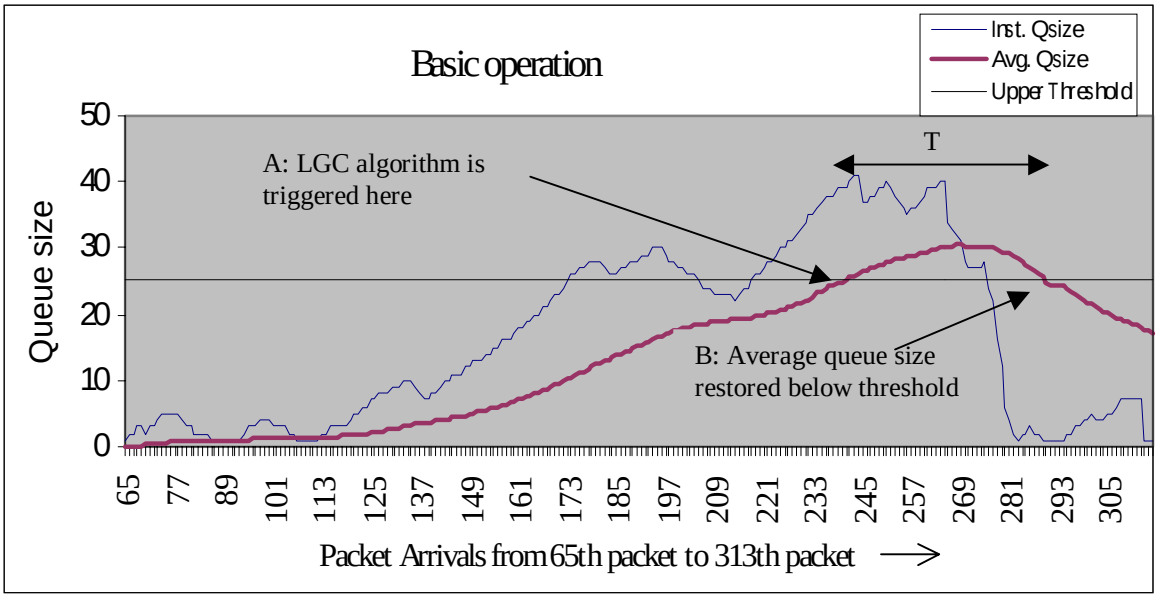

Figure 5.3 Plot of queue size v/s packet arrivals for Node 7

From the nodes packet queue we observe that the total queue occupancy is 39 packets. Of these 21 packets belong to connection 1, 4 packets belong to connection 2, 5 packets belong to connection 3, 3 packets belong to connection 4 and 2 packets each to belong connections 5,6 and 7. Totally there are seven active connections at node 7. Fair queue occupancy is 39/7 = 5.57. With a permissible factor k of $\log_e(21)$, the permissible queue occupancy is $\lceil 5.57 * 3.0445 \rceil = 17$ packets. Connection 1 had 21 packets in the node queue and was correctly identified as an unresponsive connection. Since Node 7 is the first-hop node for this connection, the migration of the packet filter was not necessary and a packet filter for connection 1 was installed at Node 7 for a duration T_FirstHop(100) seconds. Subsequently all packets arriving from connection 1 were filtered out at node 7. The throughput for responsive connections was observed to be 100% after the LGC algorithm came into effect, but the greedy connection had a throughput of 53.5% due to active filtering at node 7. If the RED algorithm were implemented in its entirety, the throughput observed for the responsive sources would be lesser than 100% since the algorithm would drop all packets arriving at the node when it is severely congested. However, this technicality is overlooked in the evaluation of LGC since RED is not implemented in its entirety i.e. arriving packets at the node under severe congestion are not dropped or marked. We only wish to isolate the greedy connections and dynamically filter them to prove that the algorithm is successful.

Due to the bandwidth greedy nature of connection 1, we observe a sudden drop in the queue occupancy once this connection is filtered. This can be observed in the region of the graph just after the LGC algorithm is triggered. Eventually the queue size is controlled at point B. The time lapse (marked as T in Figure 5.3) between the LGC algorithm coming into effect (point A) and the reduction in average queue occupancy (point B) occurs due to the low pass filtering mechanism in the calculation of the average queue size. It confirms the requirement for the presence of an idle time (Tx > T) between two successive triggers of the LGC algorithm. If the LGC algorithm were not suspended for time Tx, it would be triggered multiple times since avg exceeds max_{th} for duration T, despite active filtering of the greedy connection.

The LGC algorithm must also ensure that non-greedy unresponsive connections must not be filtered. To verify this requirement we repeated the above experiment with source node 1 injecting 200 packets in 15 bursts with an inter-burst duration of 3 seconds. Node 1 now simulates a constant packet-rate source making a moderate demand on network bandwidth at the congested node 7. Figure 5.4 shows the actual and average queue sizes plotted against packet arrivals at node 7. The bursty nature of the connections causes the spikes in the value of instantaneous queue size at the intermediate node 7.

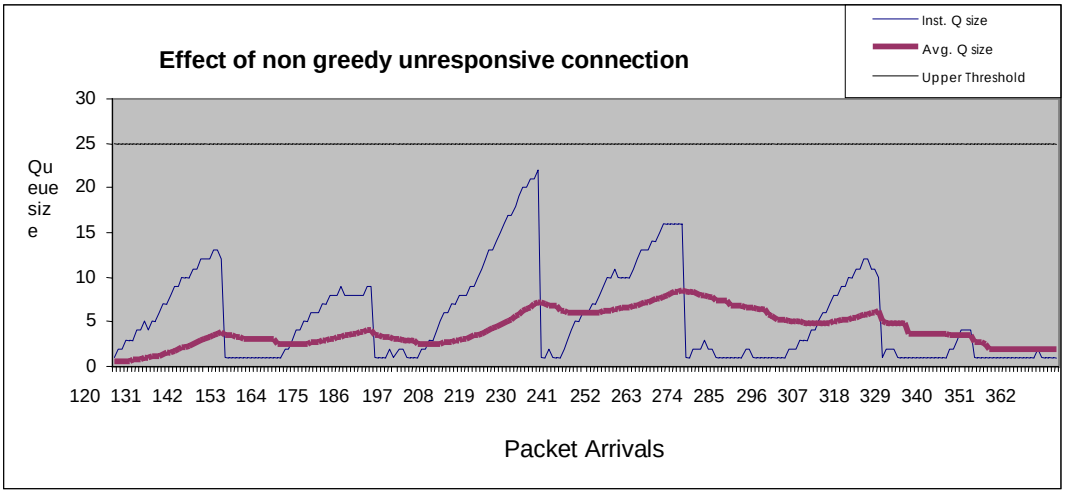

Figure 5.4 Plot of queue size v/s packet arrivals for non-greedy connections

Here, we observe that the average queue size at node 7 remains below 10 at all times implying that demand on resources does not exceed supply. Thus node 7 does not get congested and LGC is not triggered. Since queue occupancy remains low (<25), there is no packet loss and throughput is 100% for all the seven connections.

5.1.5 Experiment 2 – Mobility of the active filter

After the LGC algorithm identifies and filters a greedy connection at the congested node, it uses active messages to move the filter dynamically towards the source of the identified connection. In doing so, packets belonging to the greedy connection are filtered 'closer' to their source, thereby reducing the wastage in network resources. In this experiment we study the movement of the

mobile filter towards the source of the greedy connection and the effect of installing the mobile filter at a node. The network topology for the experiment is shown in Figure 5.5.

Node 1 is an unresponsive packet source. Nodes I1, I2 and I3 are interconnecting nodes that forward packets. Node S is the sink for all the packet sources. Nodes 2,3,4 and 5 are responsive packet sources that provide cross traffic to congest I3. Node I3 has a buffer size of 100, max_{th} set to 35 and w set to 0.02. All sources inject 250 packets with the responsive sources having an initial TCP slow-start threshold set to 32.

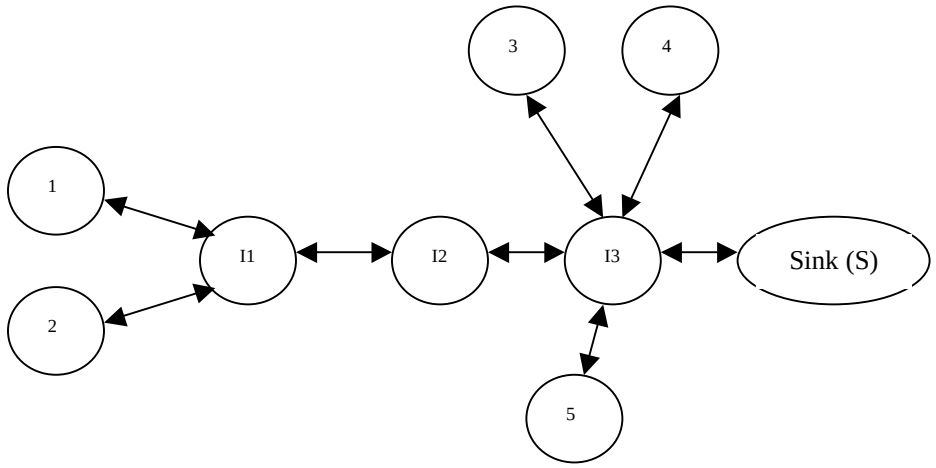

Figure 5.5 NetworkTopology for Experiment 2

In order to monitor the packet flow in the network we label the packets from the various sources as follows. Packets from source 1 are labeled a1 through a250. Packets from source 2 are labeled b1 through b250. Packets belonging to sources 3,4 and 5 are labeled similarly.

First, we consider the activities at node I3. In Figure 5.6, the x-axis shows the packets arriving at node I3 and the y-axis shows the queue size measured in packets. For brevity, the first few packet arrivals have been omitted in the chart. When the average queue size crosses 35, the LGC algorithm is triggered. At this time, I3 had received and forwarded 122 packets belonging to source 1, 84 packets belonging to source 2, 91 packets belonging to source 3, 85 packets belonging to source 4 and 66 packets belonging to source 5, making a total of 448 packets. This is shown by the dotted line in Figure 5.6.

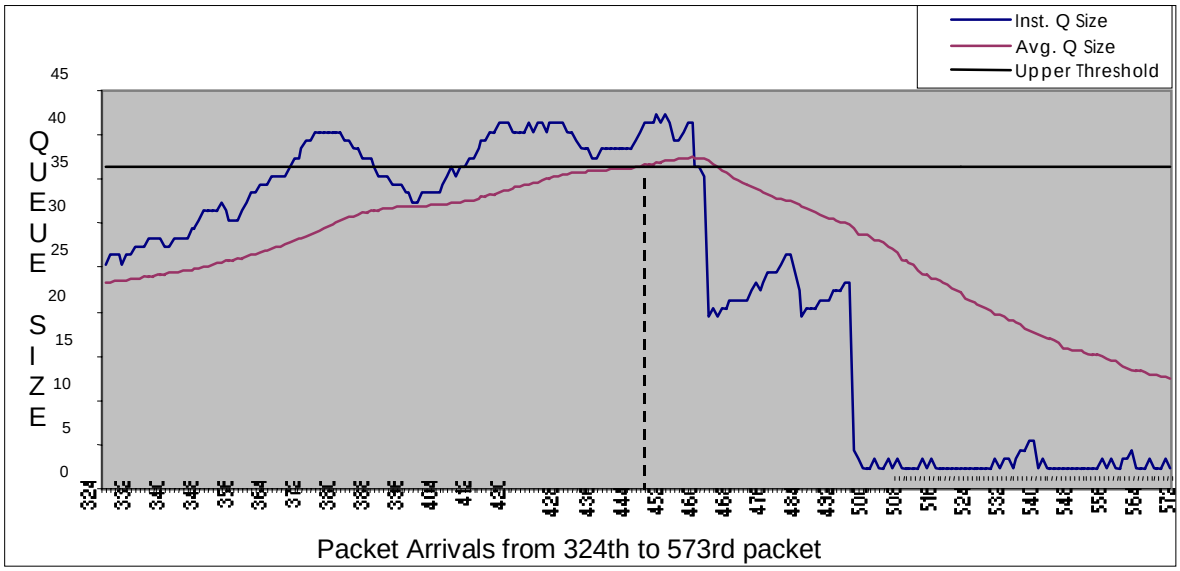

Figure 5.6 Plot of queue size v/s packet at NodeI3

Based on queue occupancy at the node, source 1 is identified as 'bandwidth greedy'. Consequently a packet filter for source 1 is installed for T_Intermediate seconds. I3 also sends an active filter message to the previous hop node I2. Now, packets belonging to source 1 are dropped at I3 as long as the packet filter remains in operation. Soon, node I2 installs a similar packet filter and the responsibility of controlling the greedy source 1 shifts one hop closer to the source. This process continued till the filter migrates to the first hop node. These actions are deduced from the packet drops for source 1 which occur successively at nodes I3 followed by I2 and finally at I1.

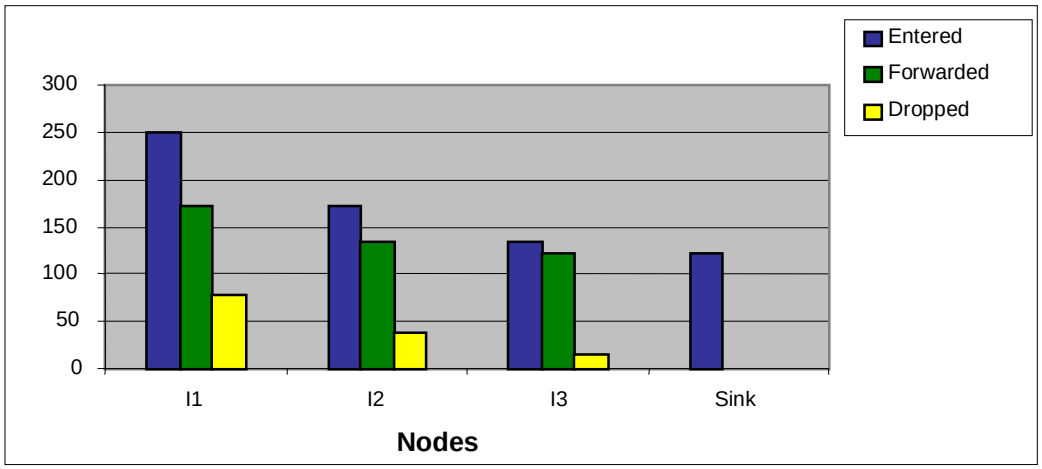

Figure 5.7 Packet flow in Experiment 2 after LGC has been triggered

In Figure 5.7, the y-axis represents the number of packets and the x-axis marks the nodes I1, I2, I3 and the sink. The bars represent the arrival and departure of packets belonging to source 1 at the nodes I1, I2 and I3. Lets start with node I3 where packet filtering begins. When the LGC algorithm was triggered, I3 had received and forwarded packets a1 through a122. It then installs the packet filter for source 1 and sends an active filter message to I2. I3 then drops packets a123 through a135 due to active filtering. Now, I2 installs a packet filter for source 1 and propagates the filter message to node I1. Subsequently I2 drops packets a136 through a173 and packets a174 through a250 were filtered at I1. Totally packets a123 through a250 are dropped after LGC is triggered.

5.1.6 Experiment 3 - Multiple bandwidth greedy connections

In this experiment we test the ability of the LGC algorithm to handle multiple bandwidth greedy connections. The network topology for this experiment is shown in Figure 5.3. Nodes 1 and 3 are greedy sources where as nodes 2, 4 and 5, 6 and 7 are unresponsive connections making a moderate demand on the network. Node 8 is the common sink for all the sources. Node 1 injects 300 packets in a single burst and node 3 injects 300 packets in 3 bursts with an inter-burst spacing of 3 seconds. The other nodes (2,4,5,6 and 7) inject 120 packets each in 30 bursts with a 2 seconds inter-burst period.

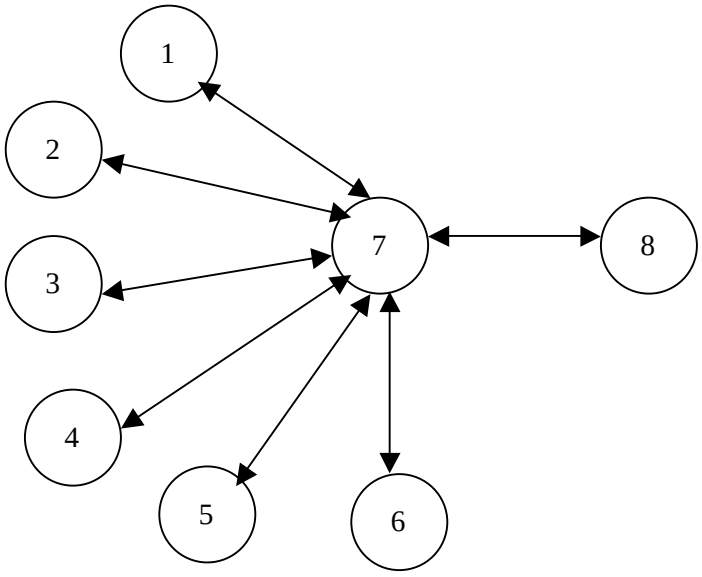

Figure 5.8 Network Topology for Experiment 3

The queue parameters for node 7 are set with queue weight = 0.04, max_{th} = 35 and buffer size = 250. A large buffer size is deliberately chosen to observe the queue occupancy at node 7 and prevent tail dropping of packets.

In Figure 5.9, the x-axis shows the packets arriving at node 7 and the y-axis shows the instantaneous and average queue sizes measured in packets. The line (y = 35) represents max_{th}. For brevity, the first 110 packet arrivals at node 7 have been omitted in the chart. Initially the average queue size remains low (<10). Once the greedy source begins injecting packets, the average queue size increases till max_{th} is crossed (point A). Now, the LGC algorithm is triggered and active filtering of greedy source 1 begins.

At this point the queue size drops but the available bandwidth is soon taken up by the second greedy source. This is observed within the Tx portion of the graph at point D. The difference here is that although the average queue size crosses the upper threshold (35 in this case), the LGC algorithm is not triggered. The reason being that a minimum time lapse of Tx is maintained between successive triggering of the LGC algorithm. After the Tx timer expires (point B), the LGC algorithm successfully identifies and filters the second greedy source. Queue occupancy is now controlled and the node emerges from its congested state (point C).

Packets arriving at the common sink node 8 reveal that throughput for the moderate connections were 100% each (mainly due to the large buffer size at node 7). Greedy connection 1 had a throughput of 57.33% and greedy connection 2 had a throughput of 46% due to active filtering at node 7.

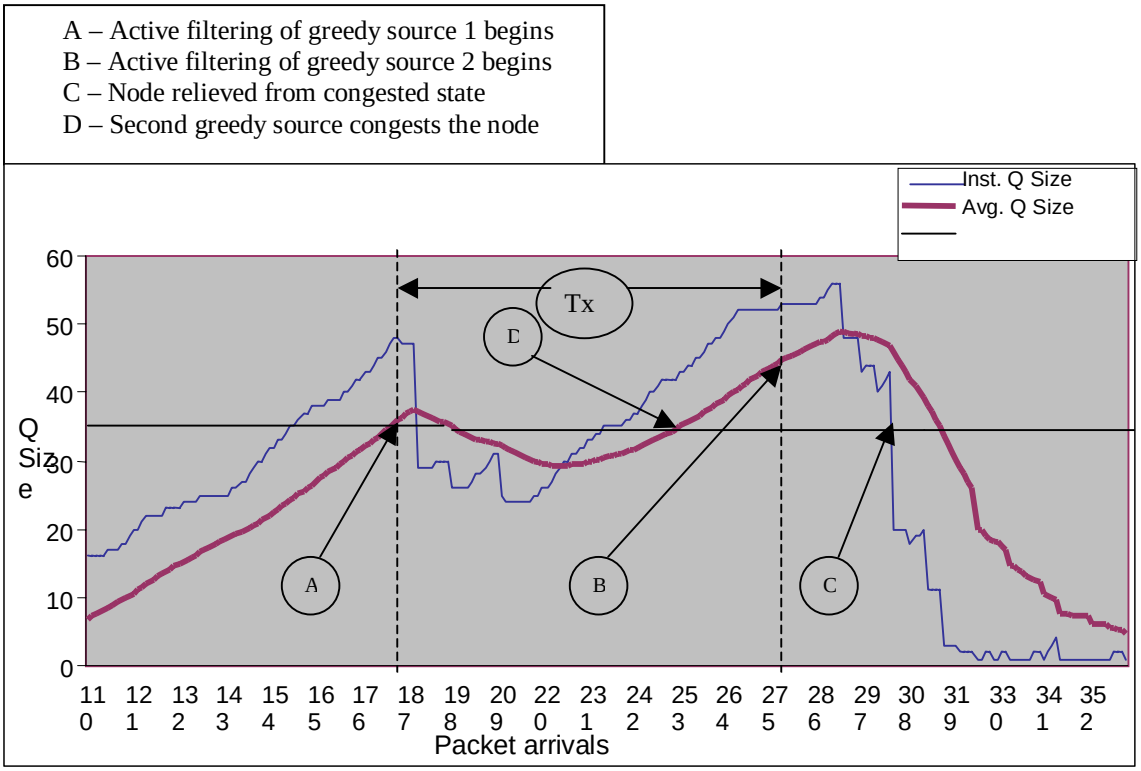

Figure 5.9 Queue size v/s packet arrivals for multiple bandwidth greedy sources

5.2 Observations of LGC

- At the congested node, if there are a large number of connections represented in the node queue, it is observed that the greedy connection tends to shut out the responsive connections and grab a large share of the bandwidth making it easier to identify greedy connections.
- There may be some cases in which multiple greedy connections compete for a limited share of the bandwidth in such a way that they restrict other responsive connections, but all of the greedy connections have individual queue occupancies within permissible limits. In this scenario the active node gets congested but the LGC algorithm fails to detect the greedy connections. For example, lets assume that there are ten connections through a node of which five are non-greedy and five are greedy sources. It is possible that when avg $\geq max_{th}$ each of the greedy sources have taken up 15% of the queue leaving the remaining 25% of the queue to be shared amongst the five responsive sources. The permissible value of queue occupancy is obtained from eq. [c] in section 4.8.1. Thus,

$$qo = 100* \log_e(3*n)/n = 100*\log_e(30)/10 = 34.01\%$$

Since all the connections including the greedy connections have queue occupancy well below this limit, the LGC algorithm does not detect them and the five responsive connections continue to receive a disproportionate share of the bandwidth. Although this example demonstrates a shortcoming of the LGC algorithm we note that such scenarios are the exception rather than the rule. It is a rare occurrence for multiple greedy connections to congest a particular node at the same time and get synchronized in such a way that they make identical demands on network bandwidth.

- The overheads of the LGC algorithm include maintaining timers and connection identifiers when the node is severely congested. However, these overheads are minimal in comparison to the benefits accrued in limiting greedy connections and relieving the node from its congested state. If extreme measures such as actively filtering out the greedy connections are not taken there is a high possibility of the node buffer being reduced to a drop-tail queue.
- If the packet filter were to be statically positioned at the congested node, the node would suffer the overhead of filtering packets at a time when its resources were scarce. Secondly, network resources such as processing time and bandwidth would be wasted between the source and the congested node at which the packets are being filtered. For the above reasons we considered it beneficial to use active network technologies to migrate the packet filter towards the source of the greedy connection and protect network resources.

Chapter 6

Conclusion and Future Work

6.1 Summary

This thesis has presented the design, implementation and evaluation of an active network architectural framework along with two popular network utilities and a mobile filtering mechanism targeted at limiting the impact of bandwidth greedy connections on congested nodes in the network. The active nodes in the network support a menu-based processing model in which end-users may select a packet processing service from an available set of possible services. The node does not maintain state as regards a particular flow but could be programmed to do so under exceptional circumstances as in the case of the LGC algorithm actively filtering packets belonging to an identified greedy connection. The active datagrams injected into the network contain a reference to the type of servicing they require and the network nodes provide this service on a best effort basis. The active nodes are implemented in Java as a user space process and execute at the application layer in the TCP/IP protocol stack. In our implementation we do not distinguish between end-nodes and intermediate nodes of the network. We have provided a user friendly GUI for configuring, operating and managing the active node.

6.2 Future Work

Our current work was focused on implementing utilities and control mechanisms on an experimental active network testbed. In this work we have borrowed many concepts from existing implementations on traditional IP networks such as Ping, Traceroute, RED and ECN. It will be interesting to pursue applications that simply cannot be supported by traditional store and forward networks; applications that must rely on intelligence within the network for their successful operation. If it can be proved that such applications significantly improve end-user satisfaction and at the same time consume network resources modestly, the active networks project is certain

to gain a wider acceptance. The future directions of this work will be in pursuing the development of such 'killer' applications. We summarize some of the future directions below:

- Designing the node to operate at the network layer will ensure that the end-to-end latency addition due to application specific packet processing within the active network will be minimized. The performance of our active network will then become comparable to traditional networks.
- Our present active network implementation assumes that the virtual links interconnecting the active nodes are bi-directional in order to reduce the complexity of the active services built on the underlying network fabric. In future implementations we wish to eliminate such simplifying assumptions so that an empirical analysis of our system will yield results that closely emulate real-world scenarios.
- A wide range of work in the development of end-user application services for active networks is currently being done. This includes Active Reliable Multicast [25], improvements in network caching [15, 32], Network Security [10], Active Bridging [12], data fission and fusion techniques within the network [37] and application oriented congestion control mechanisms [30, 33, 35]. In future implementations we wish to take a closer look at user-level application services that can directly benefit from the underlying intelligence provided by an active network.

References

1. Andrew S. Tanenbaum, *Computer Networks*, Third Edition, Prentice Hall, 1996.
2. Bernard Suter, T. V. Laxman, Dimitrios Stiliadis and Abhijit Choudhury, *Efficient Active Queue Management for Internet Routers.* IEEE/ACM Transactions on Networking, April 1988.
3. Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell and Craig Partridge, *Smart Packets for Active Networks*, BBN Technologies, 10 Moulton St, Cambridge, MA 02138.
4. B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, L. Zhang, *Recommendations on Queue Management and Congestion Avoidance in the Internet*, Request for Comments: 2309, April 1998.
5. Cui-Qing Yang and Alapati V. S. Reddy, *A Taxonomy for Congestion Control Algorithms in Packet Switched Networks.* IEEE Network Magazine July/August 1995, Volume 9, Number 5.
6. David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall and Gary J. Minden, *A Survey of Active Network Researc*h. IEEE Communications Magazine, January 1997, pp. 80-86.
7. David Wetherall, Ulana Legedza and John Guttag, *Introducing New Internet Services: Why and How.* IEEE Networks Magazine, May/June 1998.
8. David Wetherall, John Guttag and David L. Tennenhouse, *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols.* IEE OPENARCH '98, San Francisco, CA, Apfil 1998.
9. Dong Lin and Robert Morris, *Dynamics of Early Detection*. Proceedings of ACM SIGCOMM 97 Conference, Cannes, France, September 1997.
10. D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis and Jonathan M. Smith, *A Secure Active Network Architecture: Realization in SwitchWare*. IEEE Network Magazine, May/June 1998, Vol. 12 no. 3, pp. 37-45, Special Issue on Active and Controllable Networks.
11. D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles and Jonathan M. Smith, *The SwitchWare Active Network Architecture*. IEEE Network Magazine, May/June 1998, Vol. 12 no. 3, pp. 29-36, 1998. Special issue on Active and Controllable Networks.
12. D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith, *Active Bridging*. Proceedings of ACM SIGCOMM '97 Conference, Cannes, France, September 1997.
13. D. Scott Alexander, Michael W. Hicks, Angelos D. Keromytis, Jonathan T. Moore, Scott M. Nettles and Jonathan M. Smith, *A Taxonomy of Active Code*. IWAN 1999.
14. D. Tennenhouse and D. Wetherall, *Towards an Active Network Architecture*. Computer Communication Review, Vol. 26, No. 2, April 1996.
15. Edwin N. Johnson, *Using Network Level Support to Improve Cache Routing*. Master of Engineering Thesis, M.I.T., May 1998.
16. Jonathan M. Smith, D. J. Farbert, Carl A. Gunter, Scott M. Nettles, Mark E. Segal, W. D. Sincoskie, D. C. Feldmeier and D. Scott Alexander, *SwitchWare*: *Towards a 21st Century Network Infrastructure*. White Paper.
17. http://www.cc.gatech.edu/projects/canes/
18. http://www.cis.upenn.edu/~switchware/
19. http://www.cs.columbia.edu/dcc/netscript/
20. http://www.darpa.mil/ito/research/anets/

21. http://www.ir.bbn.com/projects/spkts/smtpkts-index.html
22. Information Sciences Institute, University of Southern California, *Transmission Control Protocol, Request for Comments: 793*. September 1981.
23. J. Case, M. Fedor, M. Schoffstall, J. Davin, *Simple Network Management Protocol, Request for Comments: 1157*. May 1990.
24. K. Claffy, Greg Miller and Kevin Thompson, *The Nature Of The Beast: Recent Traffic Measurements From An Internet Traffic Backbone*. From Cooperative Association for Internet Data Analysys: http://www.cetp.ipsl.fr/~porteneu/inet98/6g/6g_3.htm
25. Li-wei Lehman, Stephen J. Garland, and David L. Tennenhouse, *Active Reliable Multicast*. IEEE INFOCOM '98, San Fransisco, CA, April 1998.
26. Raj Jain, *Congestion Control in Computer Networks: Issues and Trends*. IEEE Network Magazine, May 1990, pp. 24-30.
27. Sally Floyd and Van Jacobson, *Random Early Detection Gateways for Congestion Avoidance*. IEEE/ACM Transactions on Networking, Volume 1, No. 4, pp 397-413, August 1993.
28. Sally Floyd, *TCP and Explicit Congestion Notification*. ACM Computer Communication Review, V. 24 N. 5, pp. 10-23, October 1994.
29. Sally Floyd and Kevin Fall, *Promoting the Use of End-to-End Congestion Control in the Internet*. IEEE/ACM Transactions on Networking, Volume 7, Issue 4, pp. 458-472, August 1999.
30. Samrat Bhattacharjee, Kenneth L. Calvert and Ellen W. Zegura, *On Active Networking and Congestion*. Technical Report, GIT-CC-96-02, College of Computing, Georgia Tech., Atlanta, GA, 1996.
31. Samrat Bhatacharjee, Kenneth L. Calvert and Ellen W. Zegura, *An Architecture for active Networking*. High Performance Networking (HPN '97), White Plains, NY, April 1997.
32. Samrat Bhattacharjee, Kenneth L. Calvert and Ellen W. Zegura, *Self-Organizing Wide-Area Network Caches*. Proceedings of IEEE INFOCOM '98, San Francisco, CA, March 1998.
33. Samrat Bhattacharjee, Kenneth L. Calvert and Ellen W. Zegura, *Congestion Control and Caching in CANES*. Proceedings of ICC '98, Atlanta, GA, 1998.
34. Sarah Robinson, *Multimedia Transmissions Drive Net Towards Gridlock*. The New York Times, August 23, 1999, http://www.nytimes.com/library/tech/99/08/biztech/articles/23tcp.html
35. Suresh Gopalakrishnan, Daniel Reininger and Maximilian Ott, *Framework for Packet-Based Processing of Media Flows in Networks*. Submitted to IWAN '99.
36. S. Keshav, *Congestion Control in Computer Networks*. PhD Thesis published at UC Berkeley, TR-654, September 1991.
37. Ulana Legedza, David J. Wetherall and John Guttag, *Improving the Performance of Distributed Applications Using Active Networks*. Proceedings of IEEE INFOCOM '98, San Fransisco, CA, April 1998.
38. W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.
39. W. Richard Stevens, *UNIX Network Programming, Volume 1, Second Edition: Networking APIs, Sockets and XTI*. Prentice Hall, 1998.

Appendix A

In order to compare the theoretical performance of traceroute and Atraceroute we make some basic assumptions as regards the network topology.

A.1 Assumptions:

i. Symmetrical routes between the source and destination node since it is a requirement for the successful operation of traceroute.

ii. All links that interconnect the network nodes have identical network metrics such as bandwidth, delay, etc. to aid in calculations.

iii. All nodes in the network have a constant processing time Tx, for an incoming packet. This means that the time interval between a packet entering and leaving a node is Tx and remains the same for active or passive packets.

iv. Traceroute and Atraceroute packets have the same size.

A.2 Comparison of the time complexity for traceroute and Atraceroute

Based on the above assumptions, the time taken for a packet to travel from one node to the next is a constant. Let this time be d. From assumption iii, the node resident time of the traceroute and Atraceroute packets is the same and is ignored in future calculations.

Traceroute: The first packet sent out by the source node has its TTL set to 1. The time lapse between sending out this packet and receiving a 'time exceeded' response is 2d. In general, the time lapse between sending out the nth packet and receiving the nth 'time-exceeded' response is 2(n)d. Since traceroute operates by sequentially probing and discovering nodes in the network upto the destination node, the total time (D) in discovering a node that is n hops away is

$$D = 2(d) + 2(2)d + 2(3)d + \ldots\ldots + 2(n-1)d + 2(n)d$$

$$D = (d)(n)(n+1) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [I]$$

Atraceroute: In this case only one 'originating' packet is inserted into the network and it successively probes the nodes in the network till the destination node is reached. Referring to Figure 4.3, the time taken to discover a node that is n hops away is

$$D = 2(n)d \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [II]$$

From [I] and [II] it is evident that based on identical network metrics and conditions, the time complexity for traditional traceroute is $O(n^2)$ and for Atraceroute it is $O(n)$.

A.3 Comparison of the link utilization for traceroute and Atraceroute

In the case of traceroute, the total number of links traversed by packets belonging to the traceroute utility is obtained from [I] as

$$N = n(n+1) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [III]$$

Referring to Figure 4.4, in the case of Atraceroute the total number of links traversed by the 'originating' packet is n. The 'descriptive packet' from the nth node traverses n links. Thus the total number of links traversed by the original and descriptive packets are :

$$N = n + 1 + 2 + \ldots\ldots + n$$

$$N = \tfrac{1}{2}(n)(n+3) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [IV]$$

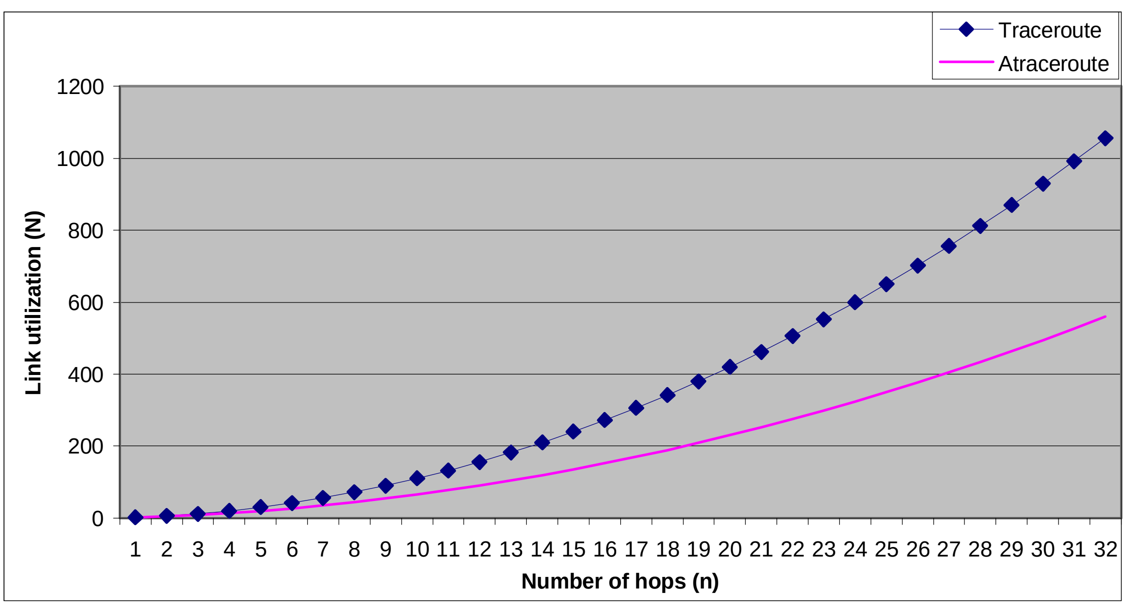

Figure A.1 Comparison of link utilization

Figure A.1 shows a plot for equations [III] and [IV]. We observe that as n increases the link utilization of Atraceroute is modest in comparison with traceroute, although both have complexity of $O(n^2)$.