T00997

Predictive Intelligent Routing: A Memory Based Reinforcement Learning Approach

★ [illegible]

Developed by:

Muhammad Adeel

Supervised by:

Dr. S Tauseef-ur-Rahman

Faculty of Applied Sciences
Department of Computer Science
International Islamic University, Islamabad
2004

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

In the name of ALMIGHTY ALLAH, The most Beneficent, the most Merciful.

Department of Computer Science,
International Islamic University, Islamabad.

17th April, 2004

Final Approval

It is certified that we have read the thesis, titled "**Predictive Intelligent Routing: A Memory Based Reinforcement Learning Approach**" submitted by **Muhammad Adeel** under University Reg. No. **63-CS/MS/02**. It is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University, Islamabad, for the Degree of **Master of Science**.

<u>Committee</u>

External Examiner
Dr. Nazir A. Sangi
Head,
Dept of Computer Science,
Allama Iqbal Open University, Islamabad.

[illegible]

Internal Examiner
Dr. Sikandar Hayat Khiyal
Head,
Dept of Computer Science,
International Islamic University, Islamabad.

[illegible]

Supervisor
Dr. S Tauseef-ur-Rahman
Head,
Department of Telecommunication Engineering,
International Islamic University, Islamabad.

[illegible]

Dedication

Dedicated to my Family who supported me in all aspects throughout my life.

A dissertation submitted to the
Department of Computer Science,
International Islamic University, Islamabad
as a partial fulfillment of the requirements
for the award of the degree of
Master of Science

Declaration

I hereby declare that this software, neither as a whole nor as a part thereof has been copied out from any source. It is further declared that I have developed this software entirely on the basis of my personal efforts made under the sincere guidance of our teachers. No portion of the work presented in this report has been submitted in support of any application for any other degree or qualification of this or any other university or institute of learning.

Muhammad Adeel
63-CS/MS/02

Acknowledgements

All praise to the Almighty Allah, the most Merciful, the most Gracious, without whose help and blessings, I would have been entirely unable to complete the project.

Thanks to my Parents who helped me during my most difficult times and it is due to their unexplainable care and love that I am at this position today.

Thanks to my project supervisor Dr. S Tauseef-ur-Rehman, whose sincere efforts helped me to complete my project successfully.

Acknowledgement is also due to my teachers and friends for their help in this project.

Muhammad Adeel
63-CS/MS/02

Project in Brief

Project Title:	Predictive Intelligent Routing: A Memory Based Reinforcement Learning Approach
Objective:	To design and implement an efficient and intelligent routing algorithm for computer networks.
Undertaken By:	**Muhammad Adeel**
Supervised By:	**Dr. Tauseef-ur-Rahman** Head, Department of Telecommunication Engineering, International Islamic University, Islamabad.
Technologies Used:	Ns-2.26, Tcl, C++, Kate editor
System Used:	Pentium® IV
Operating System Used:	Redhat Linux 9.0
Date Started:	1st September, 2003
Date Completed:	15th April, 2004

Abstract

The project aims to design and implement an efficient and intelligent routing protocol for network routing. Computer networks are becoming more and larger and complex. Different network nodes with different link characteristics and different processing power are interconnected with each other in the same network. Conventional routing algorithms are found to be increasingly incapable of handling the routing task for such networks. The need arises to develop a new algorithm which has distributed characteristics, low storage and has quick convergence. We present an efficient, distributed and intelligent routing protocol based on the framework of Reinforcement Learning for such problems. This protocol is distributed, has intelligent behavior, and has low overhead in terms of storage and processing. The whole framework is implemented on ns-2 simulator in Linux. The remainder of this document provides descriptions of the interfaces to and implementation of each of these mechanisms. In addition, a description of the design and implementation process on ns-2 simulator is also given.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

In a communication network information is transferred from one node to another as data packets. The process of sending a packet P(s; d) from its source node s to its destination node d is referred to as packet routing. Normally this packet takes multiple hops and on its way, spends some time waiting in the queues of intermediate nodes, while they are busy processing the packets that came earlier. Thus the delivery time of the packet, defined as the time it takes for the packet to reach its destination, depends mainly on the total time it has to spend in the queues of the intermediate nodes. Normally, there are multiple routes that a packet could take, which means that the choice of the route is crucial to the delivery time of the packet for any (s,d) pair. If there was a global observer with current information about the queues of all nodes in the network, it would be possible to make optimal routing decisions: always send the packet through the route that has the shortest delivery time at the moment.

1.1 State of Modern Computer Networks

Unfortunately In the real world, such complete, global information is not available, and the performance of the global observer is an upper bound on actual performance. Instead, the task of making routing decisions has to be shared by all the nodes, each using only local information. Thus, a routing policy is a collection of local decisions at the individual nodes. When a node x receives a packet P(s; d) originating at node s and destined for node d, it has to choose one of its neighboring nodes y such that the packet reaches its destination as quickly as possible.

The simplest policy is the shortest-path algorithm, which always routes packets through the path with the minimum number of hops. This policy is not always good because some intermediate nodes, falling in a popular route, might have large queues. In such cases it would be better to send the packet through another route that may be longer in term of hops but results in shorter delivery time. Hence as the traffic builds up at some popular routes, alternative routes must be chosen to keep the average packet delivery time low.

Bellman-Ford Routing (BFR) algorithm is by far the most widely used distance vector adaptive routing algorithm. In BF, each node has two tables which contain, for each possible destination,

1. A cost (cost table) or minimum delivery time for sending a packet to that destination.
2. The node's neighbor (routing table) to which the packet should be forwarded to reach the destination for the corresponding cost.

Neighboring nodes exchange their cost tables frequently for adaptation. The drawback being an enormous overhead of exploration (exchange of routing information between nodes) and a slow rate of learning.

1.2 Use of Artificial Intelligence

Reinforcement learning is a relatively new and emerging area of machine learning theory [4]. Reinforcement learning aims to develop successful techniques for learning complex strategies from limited data in a goal-directed manner. The definition of reinforcement learning given by Sutton and Barto [4] is "Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning problem". Neural Networks and genetic algorithms are used in conjunction with reinforcement learning.

It is also known as neurodynamic programming and is closely related to dynamic programming. It works well with situations where the outcome of processing is modifications to outcome of processing is modifications to the environment itself.

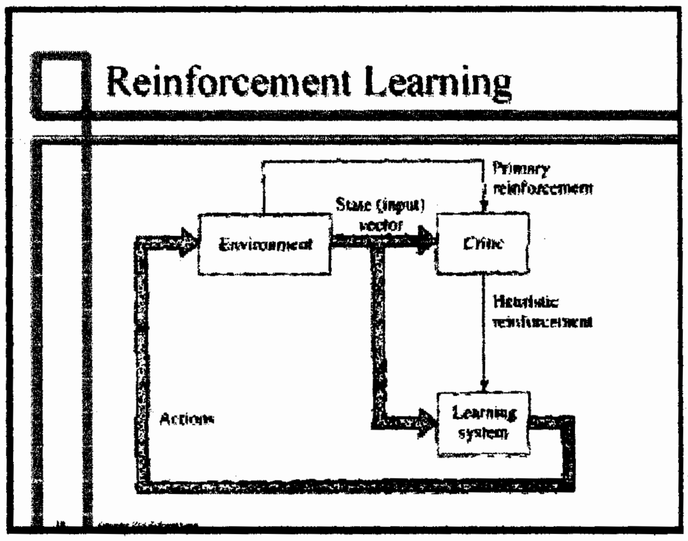


Fig 1: Reinforcement Learning

In this thesis we design and implement a distributed, efficient and intelligent protocol which provides for the development of further advanced protocols.

2. LITERATURE SURVEY

The most important part in developing any research project is Literature review. In order to understand or develop any research project we should have deep knowledge about it. We should know what research has already been done, what comments have been made by the pioneers of this field about this project and whether it is feasible to start work on it or not. For example to start a work on a network protocol, extensive literature survey about routing mechanisms, is needed. Similarly to understand an embedded operating system we should have the knowledge about the following things.

2.1 Computer Networks

With the invention of computers, new paradigms in computing begin. Computers were used to solve different problems and perform tasks for their users but soon it became obvious that a single computer, no matter how fast it was, needed to interact with other computers for different reasons [1]. Therefore computers were connected with each other through different media and at different speeds. Different protocols were written to support these interconnections.

2.1.1 Networks with static topology

These are the networks where the topology is considered to be static. No nodes goes down and no link changes its state. Such networks are very rare.

2.1.2 Networks with dynamic topology

As mentioned earlier, computers were interconnected with each other. This was no simple task. Some computers were old, had slow processing power, others had high processing speed, still others were super computers. Similarly the links between them were of varying speed; ranging from kbps to Mbps. There was a big problem as to how to connect these machines with each other. The problem was compounded by the fact that the network nodes were prone to failure. Any node could stop functioning at any time, similarly any offline node started functioning again. Complex protocols like TCP had to be written to support these connections at the software level [2].

2.2 Routing Protocols

Routing protocols are the mechanisms by which different entities in a node communicate with each other. Routing protocols handle the communication of the nodes with each other. Routing protocols provide different services to the nodes. Once two nodes have routing protocols running on them, which are interoperable with each other then they can communicate easily with each other. The design of routing protocols is not easy. Many factors have to be taken care of to design a routing protocol.

2.2.1 Classification of routing protocols

Routing protocols can be classified [1][2] into categories depending on their properties.

1. Centralized vs. Distributed.
2. Static vs. Adaptive
3. Reactive vs. Proactive

2.2.1.1 Centralized vs. Distributed

One way to categorize the routing protocols is to divide them into centralized and distributed algorithms. In centralized algorithms, all route choices are made at a central node, while in distributed algorithms, the computation of routes is shared among the network nodes.

2.2.1.2 Static vs. Adaptive

Another classification of routing protocols relates to whether they change routes in response to the traffic input patterns. In static algorithms, the route used by source-destination pairs is fixed regardless of traffic conditions. It can only change in response to a node or a link failure. This type of algorithm cannot achieve high throughput under a broad variety of traffic input patterns as explained in [1][2]. Most major packet networks use some form of adaptive routing where routes used to route between source-destination pairs may change in response to congestion [1][2].

2.2.1.3 Proactive vs. Reactive

A third classification that is more related to ad hoc networks is to classify the routing algorithms as either proactive or reactive. Proactive protocols attempt to continuously evaluate the routes within the network, so that when a packet needs to be forwarded, the route is already known and can be used immediately. The family of Distance-Vector protocols is an example of a proactive scheme. Reactive protocols, on the other hand, invoke a route determination procedure on demand only. If a route is unknown, the source node initiates a search to find one, which tends to cause a traffic surge as the query is propagated through the network. Nodes that receive the query and have a route to the requested destination respond to the query. In general, reactive protocols are primarily interested in finding any route to a destination, not necessarily the optimal route. Data sent in networks using reactive protocols do tend to suffer a delay during the search for a route. Under highly dynamic link conditions, reactive protocols are expected to generate less overhead and provide more reliable routing than proactive routing, but at the cost of finding the optimal route.

The family of classical flooding algorithms belongs to the reactive group. Proactive schemes have the advantage that when a route is needed, the delay before actual packets can be sent is very small. On the other hand, proactive schemes need time to converge to a steady state. This can cause problems if the topology is changing frequently.

2.2.2 Conventional Routing Algorithms

Because many of the proposed ad hoc routing protocols have a traditional routing protocol as underlying algorithm, it is necessary to understand the basic operation for conventional protocols like link-state, distance vector, and source routing. More detailed information about conventional routing algorithms can be founded in [1][2].

2.2.2.1 Shortest Path Routing

The idea in shortest path routing is to build a graph from the network topology, with each node of the graph representing a node and each arc of the graph representing a link. To choose a route between a given pair of nodes, the algorithm just finds the shortest path between them on the graph. One way of measuring the path length is the number of hops. Another metric is the geographic distance in kilometers or even the cost of the link. Several algorithms for computing the shortest path between two nodes of a graph are known. Under shortest path routing a packet from a given source will always take the same route. Several algorithms for computing the shortest path between two nodes of a graph are known. The best known algorithm is formed by Dijkstra [1][2][9].

2.2.2.2 Flooding

Many routing protocols use broadcast to distribute control information, that is, send the control information from an origin node to all other nodes. A widely used form of broadcasting is flooding and operates as follows. The origin node sends its information to its neighbors (in the wireless case, this means all nodes that are within transmitter range). The neighbors relay to their neighbors and so on, until the packet has reached all nodes in the network. A node will only relay a packet once and to ensure this some sort of sequence number can be used. This sequence number is increased for each new packet a node sends.

If the mobility is very high, flooding is a possible approach to route packets. The main problem is that the flooding algorithm isn't scalable to large mobile networks. Due to the large amount of nodes, excessive flooding causes the network routing protocol to break down.

2.2.2.3 Flow-based Routing

Flow-based routing is an algorithm that uses both topology and network load for routing. If, for example, there is always a huge amount of traffic at a certain link, it may be better to route the data over a longer path [10][11]. The basic idea behind this algorithm is that for a given line, if the capacity and average flow are known, it is possible to compute the mean packet delay on that line from queuing theory. From the mean delays on all the lines, it is straightforward to calculate a flow-weighted average to get the mean packet delay for the whole network. The routing problem then reduces to find the routing algorithm that produces the minimum average delay for the network. To use this technique, the network topology, information about the traffic and the capacity of the lines must be known in advance.

2.2.2.4 Distance Vector Routing

Modern computer networks generally use dynamic routing algorithms rather than the static ones described above. Two dynamic algorithms in particular, distance vector routing and link state routing are the most popular. **Distance vector routing** operates by having each router maintain a table (i.e., a vector) giving the best known distance to each destination and which line to use to get there. These tables are updated by exchanging information with the neighbors [1][2].

Compared to link-state, distance vector is more computation efficient, easier to implement and requires much less storage [1][2]. However, it is well known that distance vector can cause the formation of both short-lived and long-lived routing loops. The primary cause for this is that the nodes choose their next-hops in a completely distributed manner based on information that can be stale.

2.2.2.5 Link State Routing

The idea behind link state routing is simple and can be stated in five parts. Each router must

1. Discover its neighbours and learn their network addresses.
2. Measure the delay or cost to each of its neighbours.
3. Construct a packet telling all it has just learned.
4. Send this packet to all other routers.
5. Compute the shortest path to every other router.

Each node maintains a view of the complete topology with a cost for each link. This cost is measured experimentally. To keep these costs consistent; each node periodically broadcasts the link cost of its outgoing links to all other nodes using flooding. A node that receives this information, updates its view of the network and applies a shortest path algorithm to choose the next hop for each destination.

Some link costs in a node view can be incorrect because of long propagation delays, partitioned networks, etc. Such inconsistent network topology views can lead to formation of routing-loops.

These loops are short-lived, because they disappear in the time it takes a message to traverse the diameter of the network. Link state routing is discussed in detail in [1].

2.3 Reinforcement Learning

Reinforcement learning dates back to the early days of cybernetics and work in statistics, psychology, neuroscience, and computer science. In the last five to ten years, it has attracted rapidly increasing interest in the machine learning and artificial intelligence communities [4].

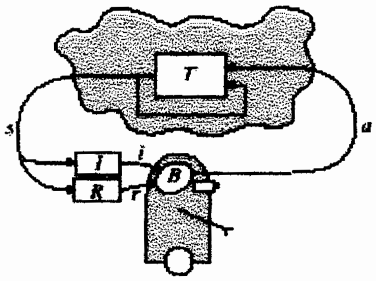


Fig 2: Standard Reinforcement Learning Model

Reinforcement learning is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment. Its promise is beguiling: a way of programming agents by reward and punishment without needing to specify how the task is to be achieved. But there are formidable computational obstacles to fulfilling the promise [4].

2.3.1 Reinforcement Learning Model

In the standard reinforcement-learning model, an agent is connected to its environment via perception and action, as depicted in Figure 2. On each step of interaction the agent receives as input, i, some indication of the current state, s, of the environment; the agent then chooses an action, a, to generate as output. The action changes the state of the environment, and the value of this state transition is communicated to the agent through a scalar reinforcement signal, r. The agent's behavior, B, should choose actions that tend to increase the long-run sum of values of the reinforcement signal. It can learn to do this over time by systematic trial and error, guided by a wide variety of algorithms.

The model is formally defined as

- A discrete set of environment states S
- A discrete set of agent actions, A;
- A set of scalar reinforcement signals typically {x|x, x is a set of real number s}
- An input function I is also defined, which determines how the agent views the environment state. Here is an example to understand the working of the reinforcement learning model.
    1. Network: You have a packet as input. You can take 4 actions. (4 paths to forward the packet)
    2. Routing Agent: I will take action 4.
    3. Network: You have a reward of 10 reinforcement learning points. Next packet as input. You can take 2 actions
    4. Routing Agent: I will take action 2
    5. Network: You have a reward of -7 reinforcment learning points. Next packet as input. You can take 3 actions.
    6. I will take action 1
    7. You have a reward of 5 reinforcement learning points. Take next packet as input. You can take 5 actions

In this way an agent learns how to use information available at run-time to decide about the actions it needs to take. It's the agent's responsibility to devise a policy to reap maximum reward (short term or long term). We assume that our environment is stationery.

Reinforcement learning theory is developed for discrete cases. When we think of optimizing our algorithms by using reinforcement learning we have to determine our model of optimality.

2.3.1.1 Finite Horizon Model

This is the simplest model. An agent at any time t should optimize its expected reward for the next h steps.

$$E\left(\sum_{t=0}^{h} r_t\right)$$

2.3.1.2 Infinite Horizon Discounted Model

The only difference this model has from the above model is that of geometrically discounting the rewards received in the future by the discount factor γ (0<γ<1)

$$E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right)$$

If the discount factor approaches 1 then infinite horizon model is reduced to Finite Horizon model.

2.3.1.3 Average Reward Model

In Average reward model the agent tries to optimize it's long term average reward.

$$\lim_{h \to \infty} E\left(\frac{1}{h}\sum_{t=0}^{h} r_t\right)$$

2.3.1.4 Bias Optimal Model

This model improves upon the average reward model. It tries to get extra reward at the start and also maximizes the long run average.

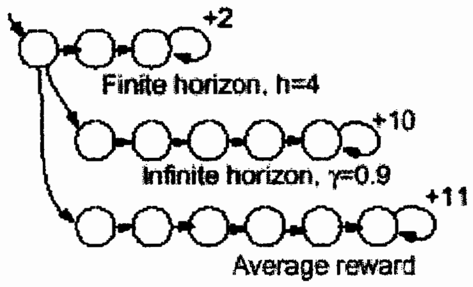


Fig 3: Comparison of Models of Optimality

2.3.2 Selection of Optimal Policy

The optimal policy depends on the model of optimality chosen. If finite horizon model is chosen then the optimal policy may be different than the policy chosen when infinite horizon model is in use.

The correct selection of optimal policy was found to be very important for the efficient use of reinforcement learning in the application.

2.3.3 Guidelines for selecting Optimal Models

Following are the guidelines for selecting the optimal models

1. When the agent's lifetime is known, finite-horizon model may be selected.
2. When the agent's lifetime is unknown, either infinite-horizon or bias optimal model may be selected.
3. If long term average reward is required then average reward model may be selected.

2.3.4 Measuring Agent Performance

1. If an agent does achieve near optimal performance but in a large amount of time

then it is useless in many applications. Speed of convergence is needed.

2. Agent performance can also be checked at specified intervals of time.

2.3.5 Application of Reinforcement Learning

Reinforcement learning is mostly applied to static environments. Only recently it's also being used in dynamic environments. Q-Learning and Q-Routing are two well known examples [7][9].

2.3.5.1 Single State k-armed bandit problem

This is a simple reinforcement learning problem. The agent is supposed to be in a room with a collection of k gambling machines. Each machine has one handle to pull. The agent can pull any handle on every turn. Pulls are allowed upto h. When handle of machine i is pulled it outputs either 1 or 0. This is called the reward or payoff of the machine. p_i is the priority associated with each machine.

This example best illustrates the tradeoff between exploration and exploitation. The agent may keep on choosing a handle with high probability of giving a reward of 1 or it may try to explore another handle with low priority also. It also depends on how long the agent plays the game.

Dynamic programming can be employed for solving this problem. A table may be made with entries for actions taken for all the machines and reward given by all of them. Then the expected payoff can be calculated and the remaining pulls may be used optimally.

2.4 Q-Learning

Recently the exploration/exploitation framework of reinforcement learning was used in Q-Routing and Q-Learning to learn optimistic routing for a dynamic network [6][7].

To learn a controller for a task, two approaches may be used. In model-based approach, the learning agent must first learn the model of the environment and then use this knowledge to learn an effective control policy for the task. In the model-free approach a controller is learned directly from the actual outcome. Reinforcement learning is an example of the model-based approach.

On-going tasks like network routing require the learning process to be continous. Some mechanism may also be adopted to access the reward after an action. This is solved by using temporal differences, a model-based approach. One such strategy is the Adaptive Heuristic Critic (AHC). It consists of two components, a critic (AHC) and a reinforcement learning controller (RL). For every action taken by the RL, there is a reward generated by the environment which is converted to a reinforcement signal for the RL by the AHC.

Q-Learning [6] is a method proposed by Watkins for solving the Markovian Decision Problems (MDP). It is a direct method because it does not use an explicit model of the dynamic system underlying the decision problem. In Q-learning the states and the possible actions in a given state are discrete and finite in number.

Q-learning solves the problem of finding the action that returns the maximum value. In case of a non-deterministic MDP, value iteration requires that we find the action that returns the maximum *expected* value (the sum of the reinforcement and the integral over all possible successor states for the given action). For example, to find the expected value of the successor state associated with a given action, one must perform that action an infinite number of times, taking the integral over the values of all possible successor states for that action. Rather than finding a mapping from states to state values (as in value iteration), Q-learning finds a mapping from state/action pairs to values (called Q-values). Instead of having an associated value function, Q-learning makes use of the Q-function. In each state, there is a Q-value associated with each action. The definition of a Q-value is the sum of the (possibly discounted) reinforcements received when performing the associated action and then following the given policy thereafter.

Likewise, the definition of an optimal Q-value is the sum of the reinforcements received when performing the associated action and then following the optimal policy thereafter. Q-learning differs from value iteration in that it doesn't require that in a given state each action be performed and the expected values of the successor states be calculated. While value iteration performs an update that is analogous to a one level breadth-first search, Q-learning takes a single step sample.

Q-Learning [6] is a technique to solve specific problems by using the reinforcement learning approach. Q-Learning is an incremental version of dynamic programming for solving multistage decision problems. In this case it is the adaptive traffic control problem. This is a model-based approach. The controllers in such an environment should adopt such routing policies so as to minimize the average packet delivery time. The controllers usually have little or no prior knowledge of the environment. Finding the optimal policy in such an environment is very difficult. Moreover the optimal routing policy may change with time as the environment is non-stationary.

An algorithm for implementing Q-Learning is as follows

```
1. Initialize (with 0's or random values) $Q(s,a)$ for all $s \in S$ and for all $a \in A(s)$
2. Repeat (for each episode)
3. Initialize $s$
4. Repeat (for each step episode):
5. Choose $a$ from $s$ using a policy derived from Q (e.g., $\epsilon$-greedy)
6. Take action $a$, observe resultant state $s'$ and the reward $r$.
        7. $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
8. $s \leftarrow s'$; until s is terminal
```

2.5 Q-Routing

The Q-Learning framework described above was used by Boyan and Littman to develop an adaptive routing algorithm [7]. Unlike the original Q-Learning algorithm, Q-Routing is distributed in the sense that each communication node has a separate controller, which does not rely on global information of the network for decision making and refinement of its routing policy. The packet routing problem can be modelled as a Markov Decision Process. The states of the MDP are represented by the nodes of the network. The actions available in a state (in a node) are represented by the neighbour-nodes to which the packet can be sent. A node's view of the state of the network is represented by the Q-values in its routing table. The complete state of the network is represented by the Q-values in all the nodes in the network. Q-Routing uses the finite horizon optimality model.

Q-routing, learns a routing policy which balances minimizing the number of "hops" a packet will take with the possibility of congestion along popular routes. It does this by experimenting with different routing policies and gathering statistics about which decisions minimize total delivery time. The learning is continual and on-line, uses only local information, and is robust in the face of irregular and dynamically changing network connection patterns and loads. A packet routing policy answers the question: to which adjacent node should the current node sends its packet to get it as quickly as possible to its eventual destination? Since the policy's performance is measured by the total time taken to deliver the packet, there is no "training signal" for directly evaluating or improving the policy until a packet finally reaches its destination.

However, using reinforcement learning (Q-learning), the policy can be updated quickly using only local information. In general, Q-values can also be used to represent the characteristics of the system based on s and a instead of the expected reinforcement. The control action, therefore, could be a function of all the Q-values in the current state. In the Q-Routing algorithm, Q-learning is used to learn the task of finding an optimal routing policy given the current state of the network. In Q-Routing, Q-learning is used to first learn a representation of the state of the network in term of Q-values and then these values are used to make control decisions. The task of Q-learning is to learn an optimal routing policy for the network. The state s in the optimization problem of the network is represented by the Q-values in the entire network. Each node x in the network represents its own view of the state of the network through its Q-table Q_x

Given this representation of the state, the action a at node x is to choose that neighbour y so that it takes minimum time for a packet destined to node d to reach its destination if sent via neighbour node y.

3. PROBLEM DOMAIN

The project aims to enhance and implement the Q-Routing protocol. As already discussed, reinforcement learning is a promising field and it is used successfully in many problems. Also it does not has the drawbacks of neural networks as it does not needs the training data. Instead reinforcement learning trains itself according to the current situation. Reinforcement learning has further branches undergoing active research. Two of such branches are Q-Learning and Q-Routing. These fields have undergone intensive research. After Q-Learning framework was made Boyan and Littman designed the Q-Routing framework to apply the Q-Learning technique on network routing by using Q-Routing. The Q-Routing algorithm is based on Q-Learning. Hence it uses the Q-values and the principles of reward of reinforcement learning.

3.1 The Routing Problem

To support a user's need to send information to an intended recipient, a network must be able to route information from one user to another. For any type of transmission in a multi-node network, a transmitter must first gain access to the channel over which the information will be sent. Techniques to access channels range from contention-based to reservation-based. *Contention-based* methods [2] use packet-by-packet contention for the radio channel and can lead to conflicts between nodes wanting to transmit at the same time. Dynamic *reservation-based* methods reserve the channel for transmission of a packet or series of.packets in a message or call. Contention-based access tends to provide the best delay and throughput characteristics when traffic consists of small packets with bursty interarrival times, while reservation-based channel access tends to provide the best performance when traffic consists of large messages or streams of packets in a call.

In addition to gaining access to another user, the network must figure out a path to that user. This is the task of routing protocol. Getting to the destination may require making many hops at intermediate routers along the way. To achieve this goal, the routing algorithm must know about the topology of the network (i.e., the set of all nodes) and choose appropriate paths through it. It also has to take care to choose routes to avoid overloading some of the communication lines and routers while leaving others idle. This is the function of the routing protocol. Depending on the network topology, there could be multiple routes from a particular source to a particular destination and hence the time taken by the packet depends on the route it takes. The overall goal that emerges can be stated as:

What is the optimal route from a given source node to a given destination node in the current state of the network?

The state of the network depends on a number of network properties: the queue lengths of the nodes, the condition of the links (whether they are up or down), condition of the nodes (whether they are up or down) and so on. If there was a *central observer* that had information about the current state (i.e. the packet queue length) of all the nodes in the network, it would be possible to find the best route using the Weighted Shortest Path Routing Algorithm (Dijkstra [1][2]). Such a *central observer* does not exist in any realistic communication system. The task of making routing decisions is therefore the responsibility of the individual nodes in the network. The routing problem can be viewed as a complex optimization problem whereby each of the local routing decisions combine to yield a global routing policy. This policy is evaluated based on the average packet delivery time under the prevailing network and traffic conditions. The quality of the policy depends, in a rather complex manner, on all the routing decisions made by all the nodes. Due to the complexity of this problem, a simplified version is usually considered. Instead of a global optimal policy, one tries to find a collection of locally optimal ones.

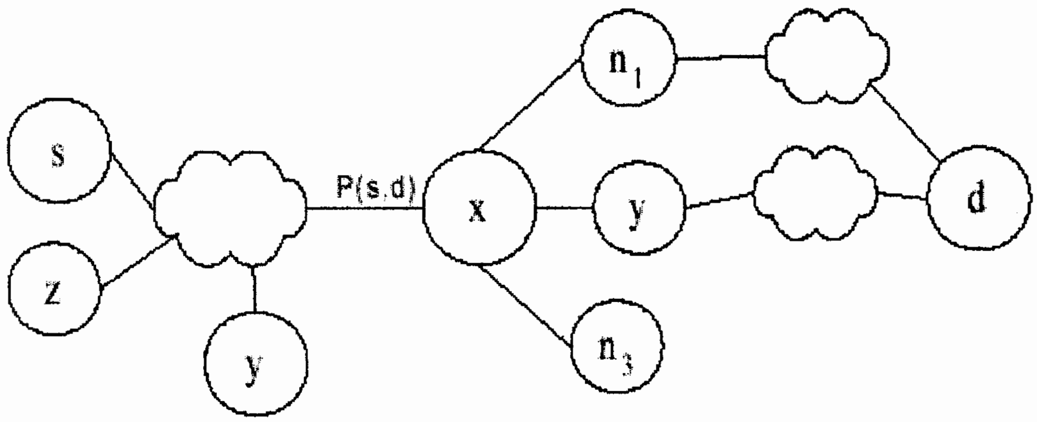


Fig 4: The Packet Routing Problem

When a node x receives a packet) , (d s P that originated at node s and is destined to node d , what is the best neighbour y of x to which this packet should be forwarded so that it reaches its destination as quickly as possible?

3.2 Project Scope

This scope of this project is described below and it includes the following things: An enhanced form of Q-Routing based on Q-Learning framework which is based on Reinforcement Learning.

1.Distributed Operation
2.Quick Convergence
3.Low processing
4.Efficient Operation
5.Low storage requirement

3.3 Objectives

The objective of this work is to enhance and implement the Q-Routing framework with the following features:

3.3.1 Distributed Operation

The algorithm will be distributed and will be running on all the nodes of the network.

3.3.2 Quick Convergence

The algorithm will be able to converge quickly after changes in the network load and topology.

3.3.3 Low Processing

The algorithm will need low processing power for its operation.

3.3.4 Efficient Operation

The algorithm will work very efficiently.

3.3.5 Low Storage Requirment

The algorithm will have very low storage requirement.

3.4 Proposed Solution

The algorithm will be implemented with the following modules.

3.4.1 Distributed Operation

The algorithm will have distributed operation i.e., it will be implemented on all the nodes of the network. On every node a local copy of the algorithm is running and taking decisions.

3.4.2 Distance Vector Routing

Distance vector routing is chosen to be the base for the Q-Routing framework. Distance vector routing was found to be efficient, had low processing requirements and quick convergence.

3.4.3 Quick Topology Discovery

The algorithm will perform quick topology discovery. This will be performed by doing broadcasting. Every node which is alive and on-line will receive the updates and send its own updates back.

4. DESIGNING

Our algorithm is designed and the simulation environment chosen for it is the NS simulator. Now let us discuss the design of each of the following:

4.1 Q-Routing

Reinforcement Learning is a relatively new field. Q-Learning is a framework based on reinforcement learning. Q-Routing is based on Q-Learning to solve the network problems.

Several distributed adaptive algorithms for efficient packet routing are proposed in the literature. The requirement for such an algorithm is that it is assumed to be implemented on a network having non uniform topology with sudden change in network traffic. The algorithm must adopt itself according to the changes and learn an efficient routing policy while satisfying the criteria of avoiding congestion and minimization of the number of hops.

We present a new distributed algorithm with adaptive behavior based on Reinforcement Learning. Traditionally algorithms based on Neural Networks and Reinforcement Learning need much more storage space and learning time than conventional algorithms which makes it unfeasible in a dynamic network with unpredictable usage patterns. The proposed algorithm uses a special Reinforcement Learning scheme which is very efficient in terms of space and time usage. The original Q-Learning algorithm used a global approach while this variation is implemented in a distributed way.

Exclusive simulations were performed to analyze this algorithm. Different network topologies with varying traffic patterns were tested. Different parameters were changed and the results were noted.

Following are the requirements for such a routing algorithm:

- Optimize end-to-end delay & bandwidth
- Balance load during peak network stress, prevent hot-spots
- Converge quickly to near-optimal paths under topology and network load changes, then revert back to original state when unloaded
- Use only local information

Q-Routing algorithm has the following characteristics:

- Q-Routing chooses best-known-delay paths through a network by recording delay values.
- Q-Routing is based on distance vector routing.
- Q-Routing extensively uses the framework of reinforcement learning
- Q-Routing is able to converge quickly
- Q-Routing is distributed.

Q-Routing was simulated on NS simulator:

- Added Q-Routing as an *rtProto* routing object
- Modified the DV and DropTail framework to support packet delay and bandwidth monitoring

Reinforcement Learning is the approach applied in the algorithm presented in this paper. By using this approach, with the help of only local information an efficient routing policy can be achieved. It is presented as follows.

Every node maintains information about other nodes in the network. Therefore when a node has to send a packet to a distant neighbor, it already has information about the next hop node which has the shortest delivery time to the destination. The next hop returns back its estimate for the destination which is used to update the policy by the current node.

4.2 Simulation Environment

NS is an object-oriented simulator. Developed at UC Berkeley it simulates different networks, implements network protocols such as TCP,UDP, traffic source behavior such as FTP, Telnet, CBR, routing algorithms such as Dijkstra and more. It also implements multicasting and some MAC layer protocols. It uses a unique split programming concept for providing its functionality. The two languages which ns uses are C++ and otcl (an extension of the popular tcl scripting language). Correspondingly there are two class hierarchies in ns.

- Compiled Hierarchy
- Interpreted Hierarchy

Both these hierarchies are closely related with each other. OTcl is the object oriented version of Tcl language and it has the same relation- ship with Tcl as C++ has with c.

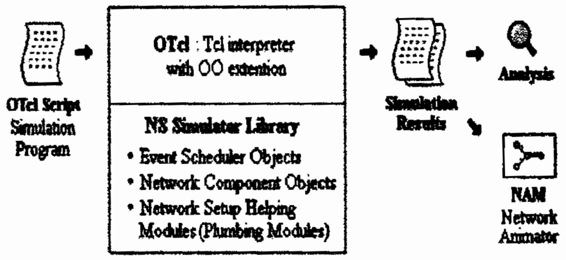


Fig 5: NS Architecture

As shown in figure 5 contains an object oriented Tcl (OTcl) interpreter that has a simulation event scheduler and network component object and network setup libraries.

T-997

NS is written in two languages C++ and OTcl. This is mainly done for efficiency reasons. The event scheduler and the basic network objects are written and compiled in C++. Through a unique binding mechanism the compiled objects of C++ are made available to the OTcl interpreter.

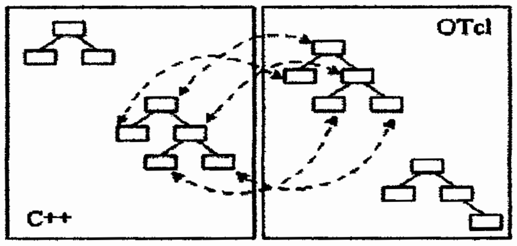


Fig 6: NS, The Duality

We can say that NS is a Object Oriented Tcl interpreter with network simulator libraries. NS has a well thought architecture. This was particularly demonstrated when the wireless/satellite support was added to NS seamlessly.

NS contains a graphical simulation display tool called NAM. NAM is used extensively to graphically visualize different simulations. It is a very capable tool and can present information such as throughput and number of packet drops at each link, although accurate simulation analysis cannot be done with that data.

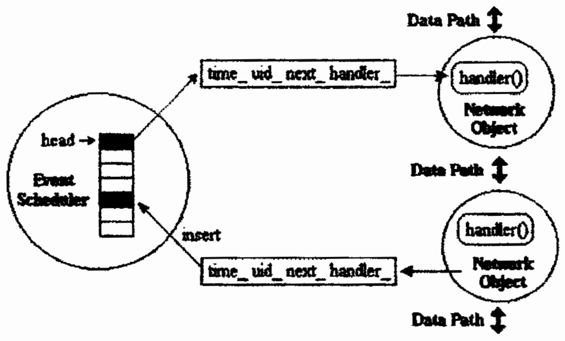


Fig 7: Internal Architecture

NS contains a discrete event scheduler. The main users of an event scheduler are network components that simulate packet-handling delay or that need timers. Above figure shows the NS event scheduler in action. Packets are send from one network object to another by using send (Packet* p) { target_->recv(p)}; for the sender and recv (Packet* , Handler* h = 0) for the receiver.

One very significant addition to the NS simulator is that of a real-time scheduler allowing NS to integrate in a real life LAN and introduce packets into it. This opens up some very exciting possibilities.

A simple NS simulation may consist of 2 nodes and a duplex link between them. A time interval may be set using a scheduler after which the nodes may start communication with each other. A stop time may also be specified. This whole simulation can be written in about 10 lines of OTCL script.

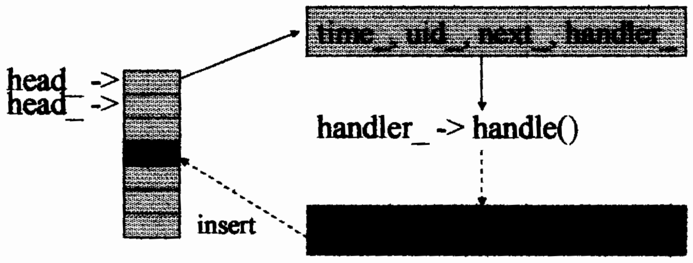


Fig 8: NS Event Scheduling

NS is a discrete event scheduler. It is not based on real-time. It has its own virtual time which is different from the virtual time. As soon as one event is executed, immediately the virtual time is advanced to next event time. The scheduler of NS is also explained in the above figure.

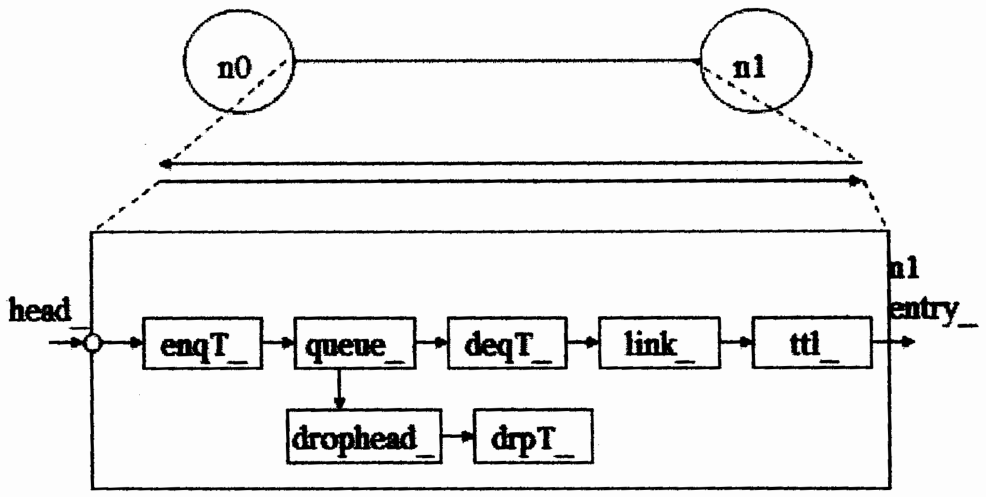


Fig 9: Internal Object Structure

As shown in figure, we can trace the queue with special trace objects. The queue is composed of several objects. The queue starts with the head_ object. At the other end of the queue the entry_ object of n1 object processes the packet.

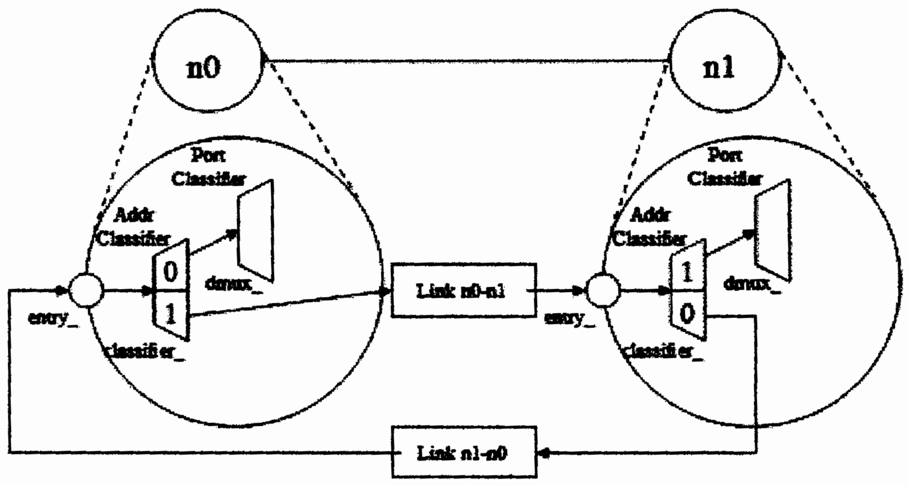


Fig 10: Internal Node Architecture

As shown in above figure the architecture of the two Q nodes is described. The two nodes have address classifiers, port classifiers and link between them. The address classifier is used to check that whether the packet is intended for this Q node or not. If yes then it sends the packet for further processing to the port classifier whose object name is dmux_. The dmux_ checks that to which port the packet may be send for further processing. The link structure between the two nodes is already described.

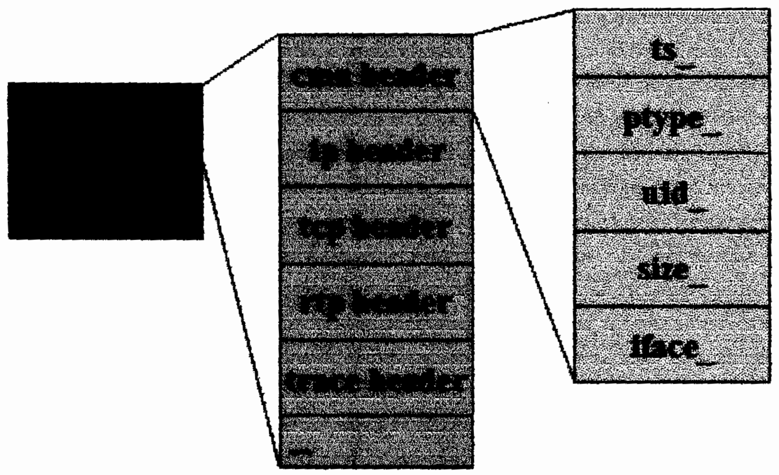


Fig 11: NS Packet Structure

NS packet header is defined in the above picture. There is a common header which is necessary in every packet. It is followed by the IP header and TCP header which are the normal headers used in TCP/IP stack. Then rtp header and trace header come followed by our custom headers. These customer headers include the header we designed: the Q-Routing header.

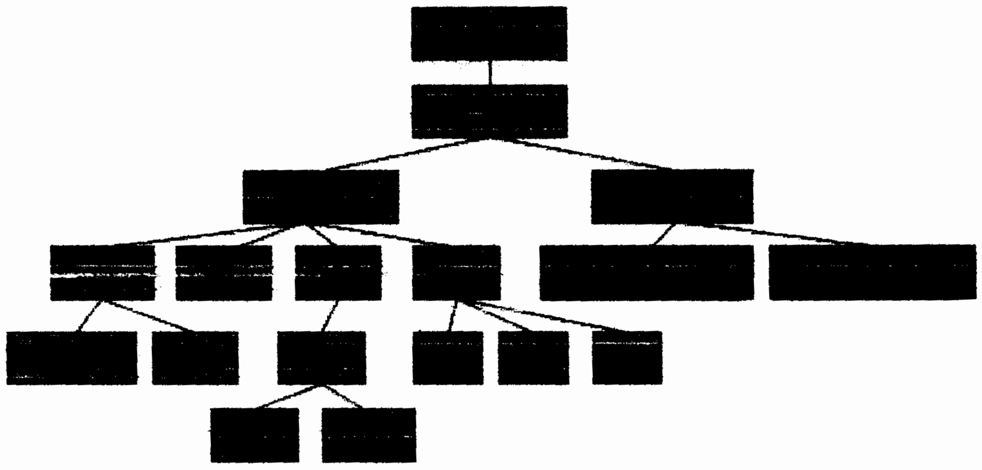


Fig 12: NS Class Hierarchy

Partial NS class hierarchy is shown in the above figure. As shown TclObject is at the top of the hierarchy immediately followed by the NsObject class.

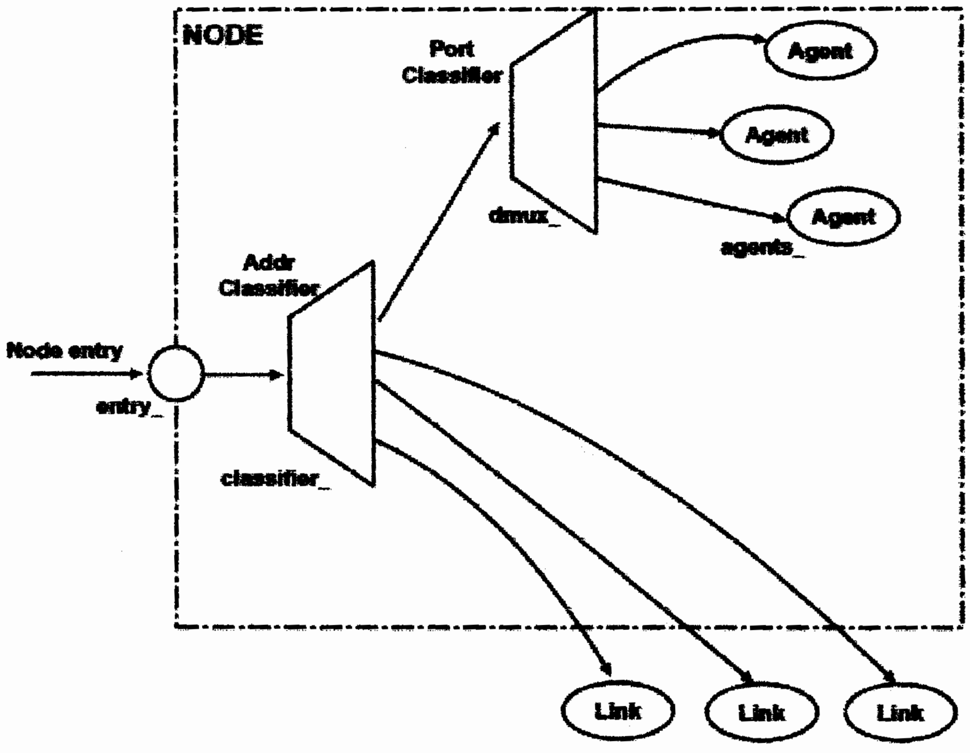


Fig 13: Node Classifiers

4.3 Modifications in NS to implement Q-Routing

Ns simulator was modified to implement Q-Routing. A new Q protocol was added. The protocol was based on distance vector routing. A new packet header entry was placed in ns/tcl/lib/ns-packet.tcl. Similarly for addition in the c++ source a new entry was placed in the ns/common/packet.h. In two other files (.cc and .h) the main source of the algorithm was written. The entries in the ns-packet.tcl and packet.h were made to make the Q-protocol visible in the c++ and the Tcl namespace. The Makefile had to be modified also to ensure that the compiler and linker knows about the new written files.

Once the required changes were done, make command was issued in the main ns folder. This resulted in the parsing of the Makefile and the recompilation of the NS simulator. Once this step was done successfully the Q-routing protocol classes became visible in the Tcl and c++ namespace. Sample simulations were run to check to correct installation of the algorithm and they worked perfectly.

5. Development and Testing

This chapter describes the development phase and the testing of the embedded operating system kernel.

5.1 Development of the Q-Routing Framework

Many modifications in NS were made to implement the Q-Routing framework. They are analyzed one by one. First the changes in the ns-default file are analyzed.

5.1.1 ns-default.tcl

This file stores the default settings for different protocols. We also add the entry for the Q-Routing protocol in this file.

```
# Dynamic routing defaults
Agent/rtProto set preference_ 200          ;# global default preference
Agent/rtProto/Direct set preference_ 100
Agent/rtProto/DV set preference_     120
Agent/rtProto/DV set INFINITY               [Agent set ttl_]
Agent/rtProto/DV set advertInterval   2
Agent/rtProto/Q set preference_      110
Agent/rtProto/Q set INFINITY                [Agent set ttl_]
Agent/rtProto/Q set advertInterval    2
```

The Agent/rtProto/Q specifies the protocol class in Tcl syntax, while the set command sets the values of its variables. As shown above the values of Agent/rtProto/Q were set. The preference_ variable is used to set the preference of the Q-routing protocol while INFINITY as set to a large value. This was done due to the fact that Q-routing is based on distance vector routing. The advertInterval_ variable was set to 2, so that the routing updates may be send to each other after interval of 2.

5.1.2 ns-packet.tcl

In this file the packet name of the new packet which the Q-Protocol will use is given. It also includes descriptions of other packets already included in NS. The new packet should be added to this file to make it visible in the Tcl namespace.

```
foreach prot {
rtProtoQ
} {
        add-packet-header $prot
}
```

As evident from the loop, we add our entry i.e., rtProtoQ in the list.

5.1.3 ns-route.tcl

In this file the function RouteLogic is passed the protocol name and the list of arguments. Suppose that in our case the proto is Q then we pass Q as protocol name and list of arguments as the no of nodes running this protocol.

RouteLogic instproc register {proto args} {

This function is an instproc and it will perform the task of initializing the protocol.

RouteLogic instproc configure {} {

In this function additional configuration of the protocol is performed.

RouteLogic instproc lookup { nodeid destid } {

This is a simple lookup function getting two parameters nodeid and destid returning the result.

Simulator instproc rtproto {proto args} {

This function is implemented by the Simulator class to support the routing protocol.

5.1.4 packet.h

In ns/common/packet.h entries were made to add the packet name in the c++ namespace.

```
enum packet_t {
```

```
PT_TCP,
PT_UDP,
PT_CBR,
PT_AUDIO,
PT_VIDEO,
PT_ACK,
PT_START,
PT_STOP,
PT_PRUNE,
PT_GRAFT,
PT_GRAFTACK,
PT_JOIN,
PT_ASSERT,
PT_MESSAGE,
PT_RTCP,
PT_RTP,
PT_RTPROTO_DV,
PT_RTPROTO_Q,
PT_CtrMcast_Encap,
PT_CtrMcast_Decap,
PT_SRM,
/* simple signalling messages */
PT_REQUEST,
PT_ACCEPT,
PT_CONFIRM,
PT_TEARDOWN,
PT_LIVE,      // packet from live network
PT_REJECT,

PT_TELNET, // not needed: telnet use TCP
```

```
PT_FTP,
PT_PARETO,
PT_EXP,
PT_INVAL,
PT_HTTP,

/* new encapsulator */
PT_ENCAPSULATED,
PT_MFTP,

/* CMU/Monarch's extnsions */
PT_ARP,
PT_MAC,
PT_TORA,
PT_DSR,
PT_AODV,
PT_IMEP,

// RAP packets
PT_RAP_DATA,
PT_RAP_ACK,

PT_TFRC,
PT_TFRC_ACK,
PT_PING,

// Diffusion packets - Chalermek
PT_DIFF,

// LinkState routing update packets
```

```
        PT_RTPROTO_LS,

        // MPLS LDP header
        PT_LDP,

        // GAF packet
      PT_GAF,

        // ReadAudio traffic
        PT_REALAUDIO,

        // Pushback Messages
        PT_PUSHBACK,

#ifdef HAVE_STL
        // Pragmatic General Multicast
        PT_PGM,
#endif //STL

        // LMS packets
        PT_LMS,
        PT_LMS_SETUP,

        // insert new packet types here
        PT_NTYPE // This MUST be the LAST one
};

class p_info {
public:
        p_info() {
                name_[PT_TCP]= "tcp";
```

```
name_[PT_UDP]= "udp";
name_[PT_CBR]= "cbr";
name_[PT_AUDIO]= "audio";
name_[PT_VIDEO]= "video";
name_[PT_ACK]= "ack";
name_[PT_START]= "start";
name_[PT_STOP]= "stop";
name_[PT_PRUNE]= "prune";
name_[PT_GRAFT]= "graft";
name_[PT_GRAFTACK]= "graftAck";
name_[PT_JOIN]= "join";
name_[PT_ASSERT]= "assert";
name_[PT_MESSAGE]= "message";
name_[PT_RTCP]= "rtcp";
name_[PT_RTP]= "rtp";
name_[PT_RTPROTO_DV]= "rtProtoDV";
name_[PT_RTPROTO_Q]="rtProtoQ";
name_[PT_CtrMcast_Encap]= "CtrMcast_Encap";
name_[PT_CtrMcast_Decap]= "CtrMcast_Decap";
name_[PT_SRM]= "SRM";

name_[PT_REQUEST]= "sa_req";
name_[PT_ACCEPT]= "sa_accept";
name_[PT_CONFIRM]= "sa_conf";
name_[PT_TEARDOWN]= "sa_teardown";
name_[PT_LIVE]= "live";
name_[PT_REJECT]= "sa_reject";


name_[PT_TELNET]= "telnet";
name_[PT_FTP]= "ftp";
name_[PT_PARETO]= "pareto";
```

```
        name_[PT_EXP]= "exp";
        name_[PT_INVAL]= "httpInval";
        name_[PT_HTTP]= "http";
        name_[PT_ENCAPSULATED]= "encap";
        name_[PT_MFTP]= "mftp";
        name_[PT_ARP]= "ARP";
        name_[PT_MAC]= "MAC";
        name_[PT_TORA]= "TORA";
        name_[PT_DSR]= "DSR";
        name_[PT_AODV]= "AODV";
        name_[PT_IMEP]= "IMEP";

        name_[PT_RAP_DATA] = "rap_data";
        name_[PT_RAP_ACK] = "rap_ack";

        name_[PT_TFRC]= "tcpFriend";
        name_[PT_TFRC_ACK]= "tcpFriendCtl";
        name_[PT_PING]="ping";

        /* For diffusion : Chalermek */
        name_[PT_DIFF] = "diffusion";

        // Link state routing updates
        name_[PT_RTPROTO_LS] = "rtProtoLS";

        // MPLS LDP packets
        name_[PT_LDP] = "LDP";

        // for GAF
    name_[PT_GAF] = "gaf";

        // RealAudio packets
```

```
                name_[PT_REALAUDIO] = "ra";

                //pushback
                name_[PT_PUSHBACK] = "pushback";

#ifdef HAVE_STL
                // for PGM
                name_[PT_PGM] = "PGM";
#endif //STL

                // LMS entries
                name_[PT_LMS]="LMS";
                name_[PT_LMS_SETUP]="LMS_SETUP";

                name_[PT_NTYPE]= "undefined";
        }
        const char* name(packet_t p) const {
                if ( p <= PT_NTYPE ) return name_[p];
                return 0;
        }
        static bool data_packet(packet_t type) {
                return ( (type) == PT_TCP || \
                        (type) == PT_TELNET || \
                        (type) == PT_CBR || \
                        (type) == PT_AUDIO || \
                        (type) == PT_VIDEO || \
                        (type) == PT_ACK \
                        );
        }
private:
        static char* name_[PT_NTYPE+1];
};
extern p_info packet_info; /* map PT_* to string name */
//extern char* p_info::name_[];

#define DATA_PACKET(type) ( (type) == PT_TCP || \
                        (type) == PT_TELNET || \
                    (type) == PT_CBR || \
                    (type) == PT_AUDIO || \
                    (type) == PT_VIDEO || \
                    (type) == PT_ACK \
                    )

#define OFFSET(type, field) ((int) &((type *)0)->field)
```

In the above file, entry was made in the enumerated structure packet_t. The entry was made by the name of PT_RTPROTO_Q. This specifies that this is a packet of a routing protocol whose name is Q. Then in class p_info a name array is used to store the enumerated values and their string equivalents. In this case following were the changes.

```
name_[PT_RTPROTO_Q]="rtProtoQ";
```

5.1.5 rtProtoQ.h

This is the main file in which the Q-Routing protocol is defined. It contains the normal C declarations, then the necessary header files are included and then the header necessary for the protocol implementation are included. Then the header hdr_Q is defined which contains the variable definitions necessary to define the header of the protocol. The static variable offset_ is defined which is required to access the offset at which the data of the Q-Routing protocol is defined. Another variable u_int32_t is also defined which is used as a metrics variable identifier.

```
#ifndef ns_rtprotoq_h
#define ns_rtprotoq_h

#include "packet.h"
#include "ip.h"

struct hdr_Q {
        u_int32_t mv_;                          // metrics variable identifier

        static int offset_;
        inline static int& offset() { return offset_; }
        inline static hdr_Q* access(const Packet* p) {
                return (hdr_Q*) p->access(offset_);
        }

        // per field member functions
        u_int32_t& metricsVar() { return mv_; }
};

class rtProtoQ : public Agent {
public:
        rtProtoQ() : Agent(PT_RTPROTO_Q) {}
        int command(int argc, const char*const* argv);
        void sendpkt(ns_addr_t dst, u_int32_t z, u_int32_t mtvar);
        void recv(Packet* p, Handler*);
};

#endif
```

As obvious from the description, first packet.h and ip.h are included, then a new structure is defined with two member variables and associated member functions. Then public class rtProtoQ is implemented, which has the necessary member functions defined.

5.1.6 rtProtoQ.cc

```
#include "agent.h"
#include "rtProtoQ.h"

int hdr_Q::offset_;

static class rtQHeaderClass : public PacketHeaderClass {
public:
        rtQHeaderClass() : PacketHeaderClass("PacketHeader/rtProtoQ",
                                        sizeof(hdr_Q)) {
                bind_offset(&hdr_Q::offset_);
        }
} class_rtProtoQ_hdr;

static class rtProtoQclass : public TclClass {
public:
        rtProtoQclass() : TclClass("Agent/rtProto/Q") {}
        TclObject* create(int, const char*const*) {
                return (new rtProtoQ);
        }
} class_rtProtoQ;


int rtProtoQ::command(int argc, const char*const* argv)
{
        if (strcmp(argv[1], "send-update") == 0) {
                ns_addr_t dst;
                dst.addr_ = atoi(argv[2]);
                dst.port_ = atoi(argv[3]);
                u_int32_t mtvar = atoi(argv[4]);
                u_int32_t size  = atoi(argv[5]);
                sendpkt(dst, mtvar, size);
                return TCL_OK;
        }
        return Agent::command(argc, argv);
}

void rtProtoQ::sendpkt(ns_addr_t dst, u_int32_t mtvar, u_int32_t size)
{
        daddr() = dst.addr_;
        dport() = dst.port_;
        size_ = size;
        Packet* p = Agent::allocpkt();
        hdr_Q *rh = hdr_Q::access(p);
        rh->metricsVar() = mtvar;

        target_->recv(p);
}

void rtProtoQ::recv(Packet* p, Handler*)
{
        hdr_Q* rh = hdr_Q::access(p);
        hdr_ip* ih = hdr_ip::access(p);
        Tcl::instance().evalf("%s recv-update %d %d", name(),
                                ih->saddr(), rh->metricsVar());
        Packet::free(p);
}
```

The above described class is the c++ implementation class. It creates packet header, then routing protocol class, makes visible the class name in Tcl, and implements the member methods command, sendpkt and receive. These functions were implemented in c++ only to make the processing faster. This is because Tcl is interpreted and therefore considerable slower than c++.

We have presented a distributed, adaptive algorithm with no global information of network topology. The algorithm succeeds in finding an efficient routing policy.

5.2 Testing

After development and implementation the protocol was tested with different traffic scenarios and topologies. The protocol was implemented and NS simulator was recompiled and linked. After linking the following command was issued on the command line to check the correct installation of the Q-Routing protocol.

```
%set ns[new Simulator]
%Agent/rtProto info subclass
```

The command was issued to check the subclasses of the rtProto class. As expected the name of the Agent/rtProto/Q was displayed

The implementation was found to be capable of full scale development. As the protocol is fully extensible, it can be easily extended to mobile ad-hoc networks or to satellite networks. The implementation was robust enough to perform at a fast speed on my AMD Athlon 1700+ system.

After implementation, several test files were compiled and then run. The ouput in graphical form was seen through NAM, which is the network animator for the NS simulator. Following is the Tcl code of one such sample file.

The protocol was found to be a balanced one with code distributed between the Tcl and c++ files. In the Tcl files one time initialization is done while in the c++ file packet processing and other heavy processing is done.

5.2.1 Testing with FTP connections

Given is the source code of a file which contains the implementation of a network topology with ftp connection setup between the nodes. The ftp connections worked flawlessly with Q-Routing.

```
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
```

```
$ns duplex-link $n0 $n2 5Mb 2ms DropTail
$ns duplex-link $n1 $n2 5Mb 2ms DropTail
$ns duplex-link $n2 $n3 1.5Mb 10ms DropTail
# Some agents.
set udp0 [new Agent/UDP] ;# A UDP agent
$ns attach-agent $n0 $udp0 ;# on node $n0
set cbr0 [new Application/Traffic/CBR] ;# A CBR traffic generator agent
$cbr0 attach-agent $udp0 ;# attached to the UDP agent
$udp0 set class_ 0 ;# actually, the default, but...
set null0 [new Agent/Null] ;# Its sink
$ns attach-agent $n3 $null0 ;# on node $n3
$ns connect $udp0 $null0
$ns at 1.0 "$cbr0 start"
puts [$cbr0 set packetSize_]
puts [$cbr0 set interval_]
# A FTP over TCP/Tahoe from $n1 to $n3, flowid 2
set tcp [new Agent/TCP]
$tcp set class_ 1
$ns attach-agent $n1 $tcp
set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink
set ftp [new Application/FTP] ;# TCP does not generate its own traffic
$ftp attach-agent $tcp
$ns at 1.2 "$ftp start"
$ns connect $tcp $sink
$ns at 1.35 "$ns detach-agent $n0 $tcp ; $ns detach-agent $n3 $sink"
# The simulation runs for
-
.
# The simulation comes to an end when the scheduler invokes the finish{} procedure below.
# This procedure closes all trace files, and invokes nam visualization on one of the trace files.
$ns at 3.0 "finish"
proc finish {} {
global ns f nf
$ns flush-trace
close $f
close $nf
puts "running nam..."
exec nam out.nam &
exit 0
}
# Finally, start the simulation.
$ns run
```

5.2.2 Testing with Packet Drops

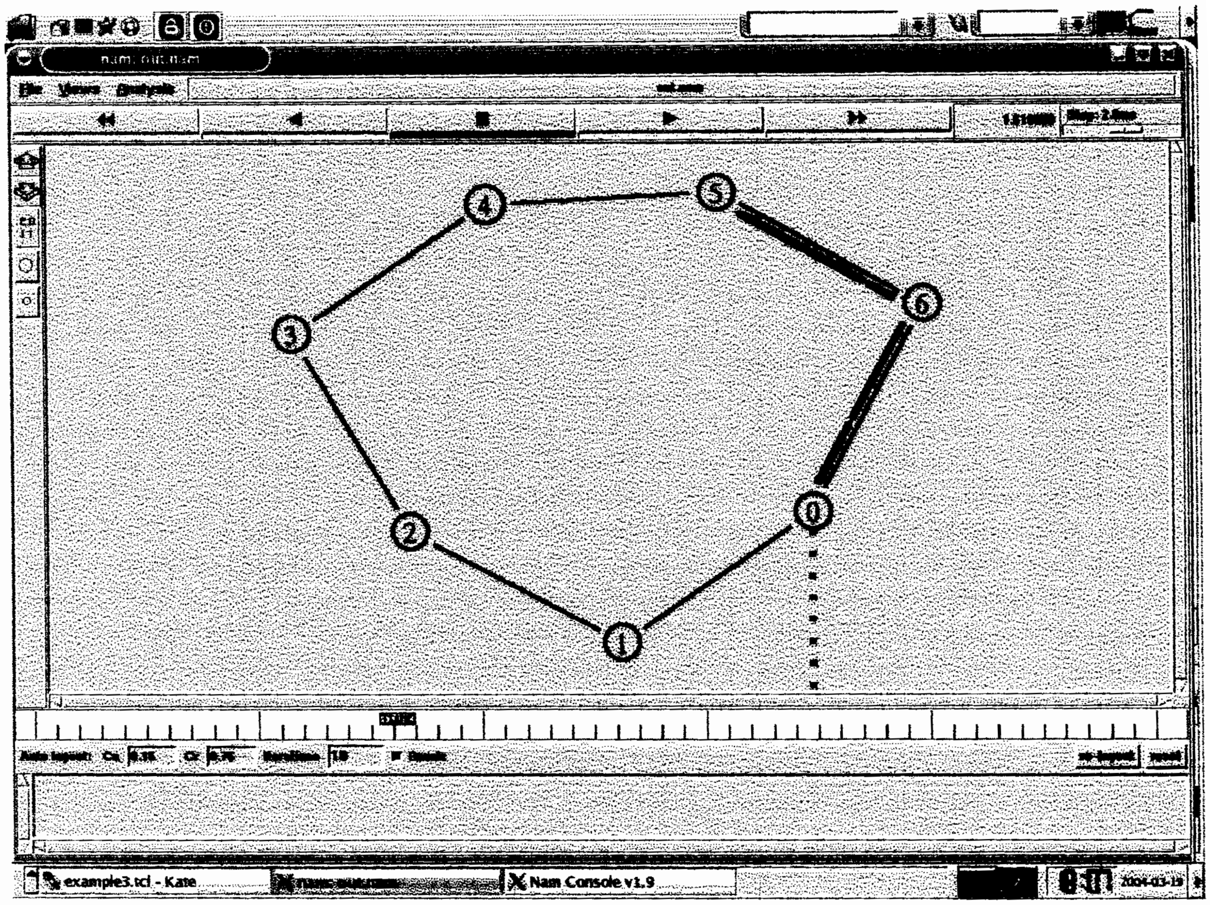


Fig 14: Testing with Packet Drops

The above figure shows the running of the protocol. Packet drop is also indicated in the figure. The topology is a simple topology consisting of 7 nodes while traffic using the Q-Routing protocol is between node 0 and node 5. APIs are also available to monitor the queue size, and packet drop information which may be used for further optimization. Classes to support such functions include

- Queue Monitoring
- Tracing
- Packet level Tracing

- Bytes Tracing
- Link Tracing
- Single Object Tracing

5.2.3 Testing with Complex Topology

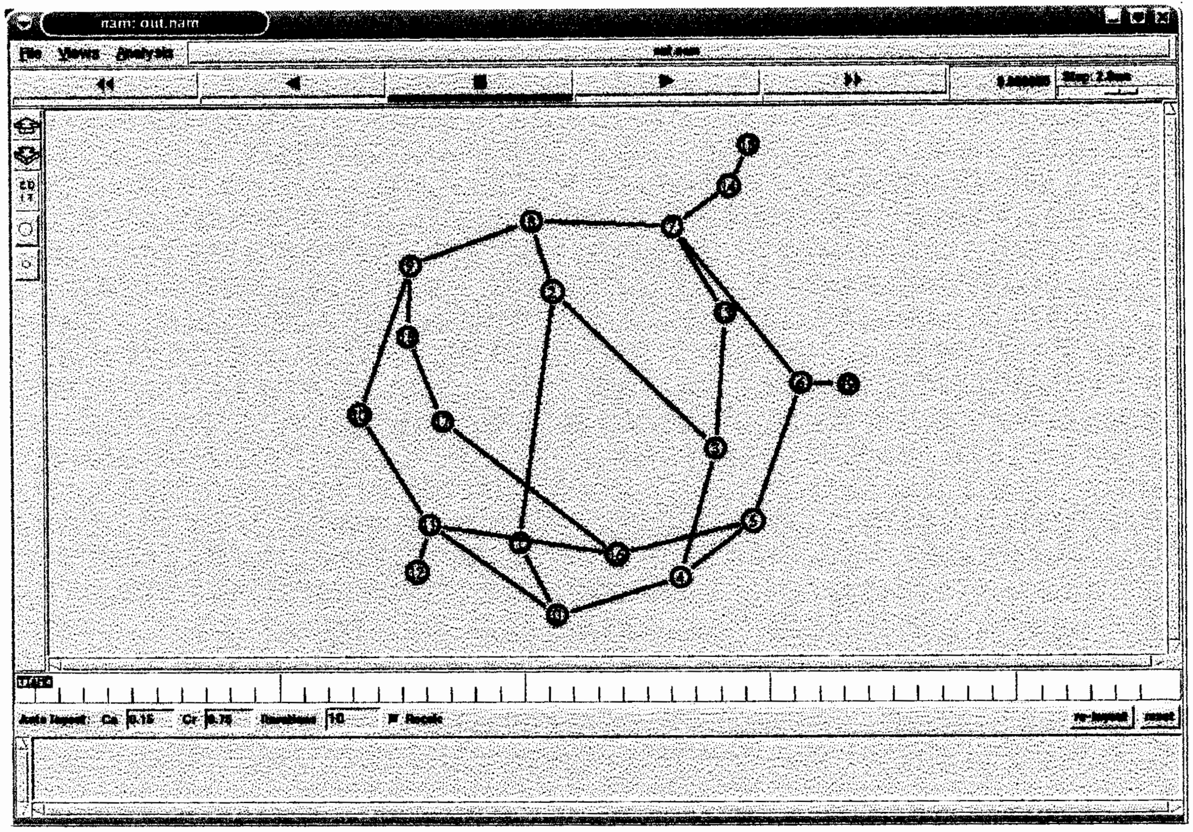


Fig 15: Testing with Complex Topology

In the above figure another topology is created. This topology is a complex one consisting of many nodes which are interconnected with each other through different links.

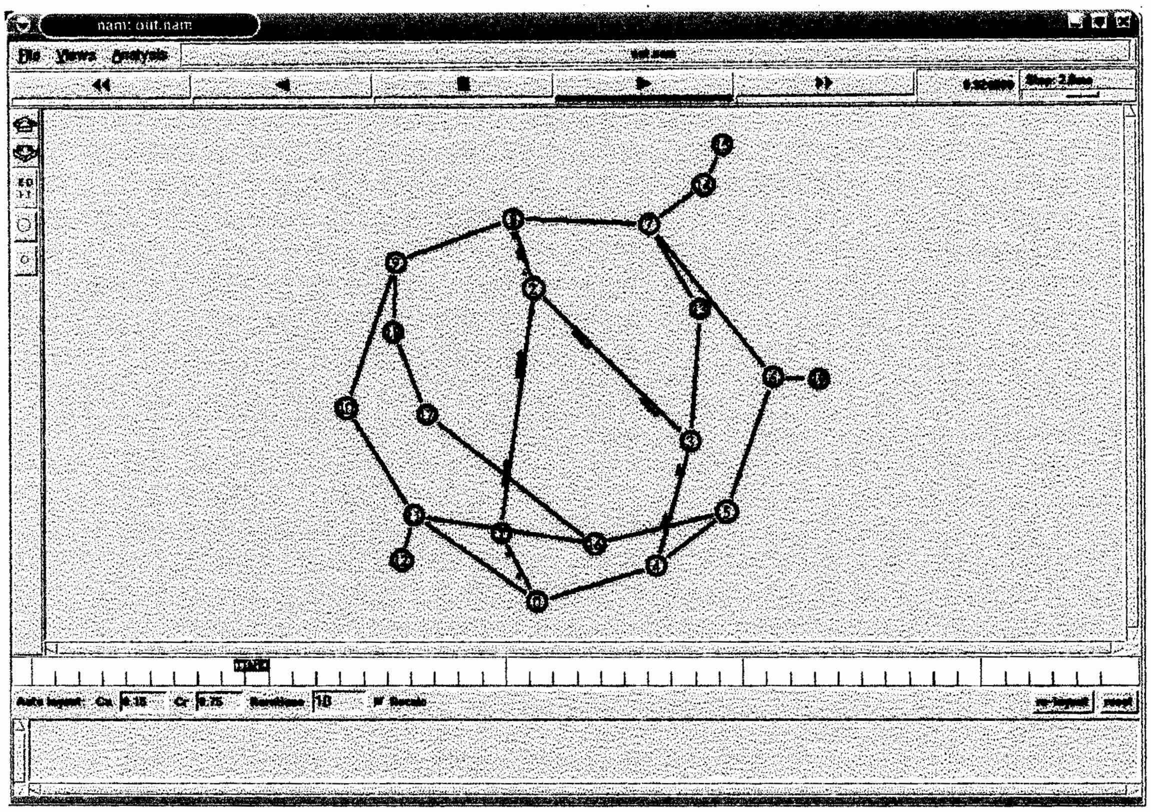


Fig 16: Testing with Complex Topology

The above figure explains the running of a complex network traffic on Q-Routing. The long black blocks are the packets which are reaching their destination by a efficient path.

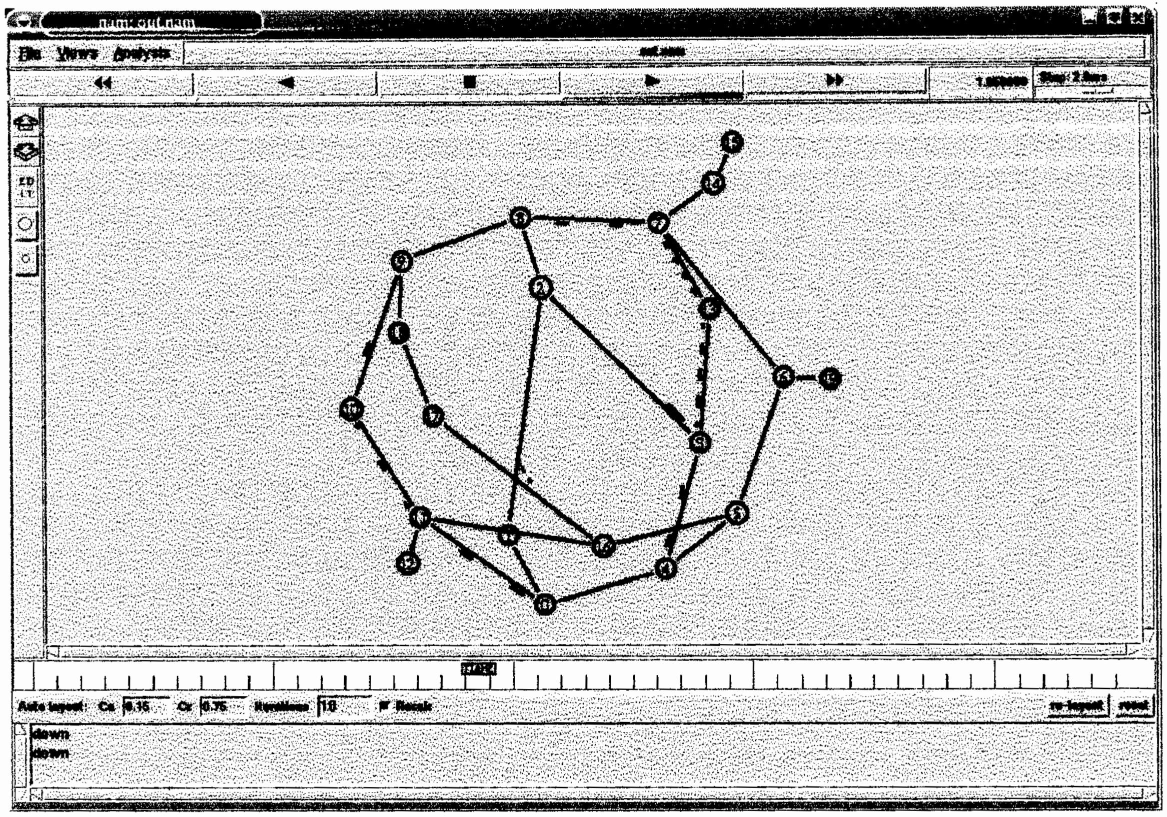


Fig 17: Testing with Complex Topology

The above figure shows the running of the network with complex topology. The red link shows that the link is currently down. Due to the efficiency of the Q-Routing protocol, the traffic will be redirected through another path. The black rectangles are the packets.

5.2.4 Testing with Multicasting

The topology given is that of nodes with multicasting support enabled. As expected the results were very encouraging.

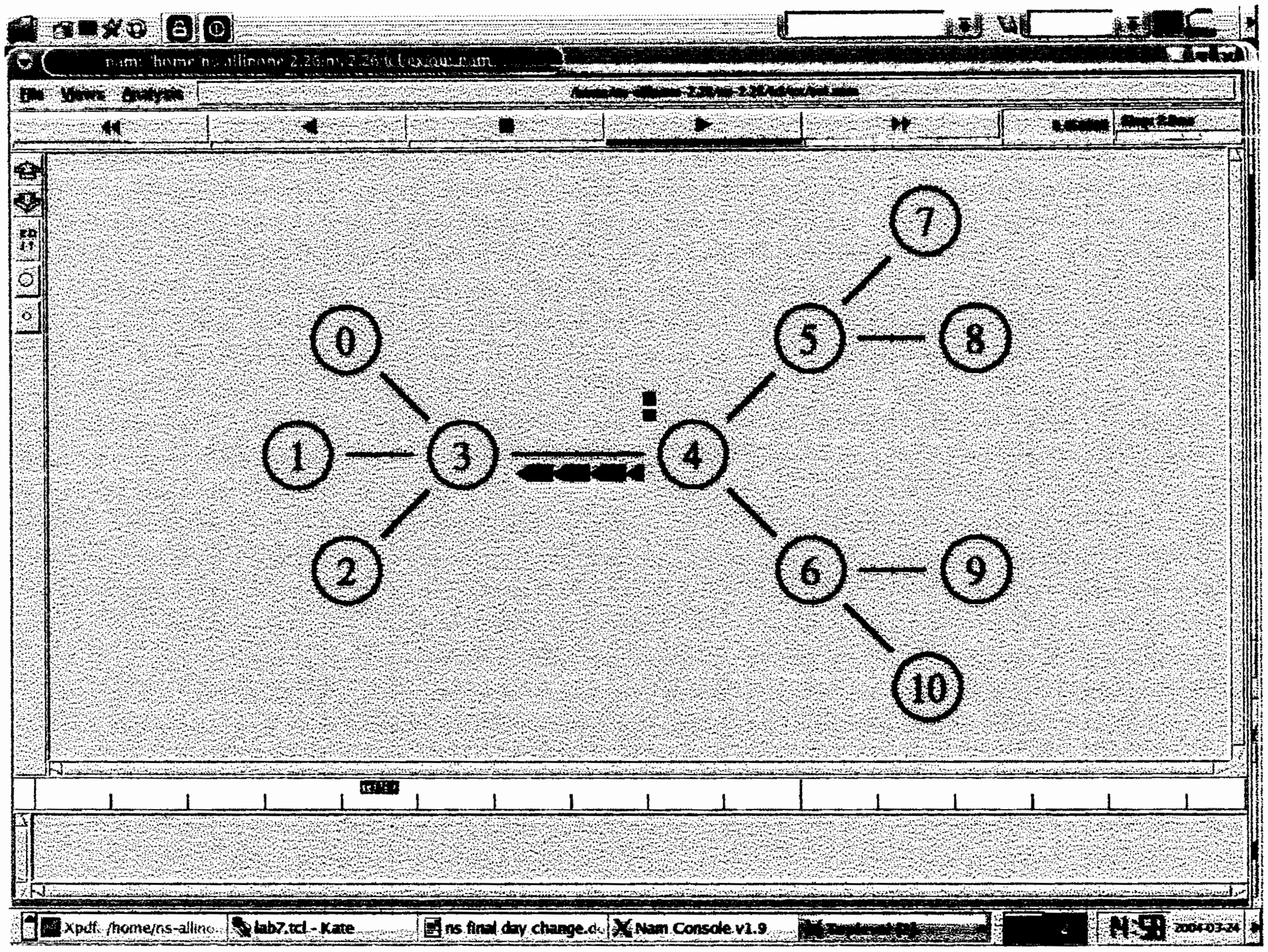


Fig 18: Testing with Multicasting

The above picture is an example of a network with ftp connections in a large topology. This topology also contains multicast support. The Q-Routing protocol is tested with multicast support. The source code is given below.

```
# Creating the core event scheduler
set ns [new Simulator -multicast on]
$ns rtproto Q
# nam coloring
$ns color 1 red
$ns color 2 blue

# Create nam trace
set nf [open out.nam w]
$ns namtrace-all $nf

# Creating 11 nodes
for {set i 0} {$i < 11} {incr i} {
   set n($i) [$ns node]
}

# Creating 10 links
# All with bandwidth 10Mb, delay 10ms, and DropTail queue
# Except the bottleneck link with bw 1.5Mb, delay 20ms, and RED queue
$ns duplex-link $n(0) $n(3) 10Mb 10ms DropTail
$ns duplex-link $n(1) $n(3) 10Mb 10ms DropTail
$ns duplex-link $n(2) $n(3) 10Mb 10ms DropTail
$ns duplex-link $n(3) $n(4) 1.5Mb 20ms RED
$ns duplex-link $n(4) $n(5) 10Mb 10ms DropTail
$ns duplex-link $n(4) $n(6) 10Mb 10ms DropTail
$ns duplex-link $n(5) $n(7) 10Mb 10ms DropTail
$ns duplex-link $n(5) $n(8) 10Mb 10ms DropTail
$ns duplex-link $n(6) $n(9) 10Mb 10ms DropTail
$ns duplex-link $n(6) $n(10) 10Mb 10ms DropTail
```

```
# Set queue limit (buffer size) to 5 packets in the bottleneck link
$ns queue-limit $n(3) $n(4) 5
$ns queue-limit $n(4) $n(3) 5

# Set link orientation for nam
$ns duplex-link-op $n(0) $n(3) orient right-down
$ns duplex-link-op $n(1) $n(3) orient right
$ns duplex-link-op $n(2) $n(3) orient right-up
$ns duplex-link-op $n(3) $n(4) orient right
$ns duplex-link-op $n(4) $n(5) orient right-up
$ns duplex-link-op $n(4) $n(6) orient right-down
$ns duplex-link-op $n(5) $n(7) orient right-up
$ns duplex-link-op $n(5) $n(8) orient right
$ns duplex-link-op $n(6) $n(9) orient right
$ns duplex-link-op $n(6) $n(10) orient right-down

# Set queue position for nam
$ns duplex-link-op $n(3) $n(4) queuePos -0.5
$ns duplex-link-op $n(4) $n(3) queuePos -0.5

# Create a TCP connection from node 9 to node 1
set tcp [new Agent/TCP]
set sink [new Agent/TCPSink]
$ns attach-agent $n(9) $tcp
$ns attach-agent $n(1) $sink
$ns connect $tcp $sink
$tcp set class_ 1
$tcp set window_ 20
# Create an FTP source and attach it to the TCP connection
set ftp [new Application/FTP]
```

```
$ftp attach-agent $tcp

# Create a TCP-friendly connection from node 7 to node 0
set tfrc [new Agent/TFRC]
set tfrcsink [new Agent/TFRCSink]
$tfrc set class_ 2
$tfrcsink set class_ 2
$ns attach-agent $n(7) $tfrc
$ns attach-agent $n(0) $tfrcsink
$ns connect $tfrc $tfrcsink

## Start the simulation
$ns at 0.2 "$ftp start"
$ns at 0.2 "$tfrc start"
$ns at 1.5 "finish"

# A finish proc to flush traces and out call nam
proc finish {} {
        global ns
        $ns flush-trace

        puts "running nam..."
        exec nam out.nam &
        exit 0
}

$ns run
```

5.2.5 Testing with Trace Files.

The file given below is a trace file. It has the standard architecture of ns. The fields denote packet type, protocol used, arrival time etc. The trace file contained the complete correct information of the network simulation.

```
+ 0 9 0 tcp 40 ------- 0 9.0 7.0 0 0
- 0 9 0 tcp 40 ------- 0 9.0 7.0 0 0
r 0.00216 9 0 tcp 40 ------- 0 9.0 7.0 0 0
h 0.00216 0 8 tcp 40 ------- 0 9.0 7.0 0 0
+ 0.00316 0 8 tcp 40 ------- 0 9.0 7.0 0 0
- 0.00316 0 8 tcp 40 ------- 0 9.0 7.0 0 0
r 0.004215 8 7 tcp 40 ------- 0 9.0 7.0 0 0
h 0.004215 7 8 ack 40 ------- 0 7.0 9.0 0 1
+ 0.005215 7 8 ack 40 ------- 0 7.0 9.0 0 1
- 0.005215 7 8 ack 40 ------- 0 7.0 9.0 0 1
r 0.00627 8 0 ack 40 ------- 0 7.0 9.0 0 1
+ 0.00627 0 9 ack 40 ------- 0 7.0 9.0 0 1
- 0.00627 0 9 ack 40 ------- 0 7.0 9.0 0 1
r 0.00843 0 9 ack 40 ------- 0 7.0 9.0 0 1
+ 0.00843 9 0 tcp 1040 ------- 0 9.0 7.0 1 2
- 0.00843 9 0 tcp 1040 ------- 0 9.0 7.0 1 2
+ 0.00843 9 0 tcp 1040 ------- 0 9.0 7.0 2 3
- 0.01259 9 0 tcp 1040 ------- 0 9.0 7.0 2 3
r 0.01459 9 0 tcp 1040 ------- 0 9.0 7.0 1 2
h 0.01459 0 8 tcp 1040 ------- 0 9.0 7.0 1 2
+ 0.01559 0 8 tcp 1040 ------- 0 9.0 7.0 1 2
- 0.01559 0 8 tcp 1040 ------- 0 9.0 7.0 1 2
r 0.017438 8 7 tcp 1040 ------- 0 9.0 7.0 1 2
h 0.017438 7 8 ack 40 ------- 0 7.0 9.0 1 4
+ 0.018438 7 8 ack 40 ------- 0 7.0 9.0 1 4
- 0.018438 7 8 ack 40 ------- 0 7.0 9.0 1 4
```

```
r 0.01875 9 0 tcp 1040 ------- 0 9.0 7.0 2 3
h 0.01875 0 8 tcp 1040 ------- 0 9.0 7.0 2 3
r 0.019493 8 0 ack 40 ------- 0 7.0 9.0 1 4
+ 0.019493 0 9 ack 40 ------- 0 7.0 9.0 1 4
- 0.019493 0 9 ack 40 ------- 0 7.0 9.0 1 4
+ 0.01975 0 8 tcp 1040 ------- 0 9.0 7.0 2 3
- 0.01975 0 8 tcp 1040 ------- 0 9.0 7.0 2 3
r 0.021598 8 7 tcp 1040 ------- 0 9.0 7.0 2 3
h 0.021598 7 8 ack 40 ------- 0 7.0 9.0 2 5
r 0.021653 0 9 ack 40 ------- 0 7.0 9.0 1 4
+ 0.021653 9 0 tcp 1040 ------- 0 9.0 7.0 3 6
- 0.021653 9 0 tcp 1040 ------- 0 9.0 7.0 3 6
+ 0.021653 9 0 tcp 1040 ------- 0 9.0 7.0 4 7
+ 0.022598 7 8 ack 40 ------- 0 7.0 9.0 2 5
- 0.022598 7 8 ack 40 ------- 0 7.0 9.0 2 5
r 0.023653 8 0 ack 40 ------- 0 7.0 9.0 2 5
+ 0.023653 0 9 ack 40 ------- 0 7.0 9.0 2 5
- 0.023653 0 9 ack 40 ------- 0 7.0 9.0 2 5
- 0.025813 9 0 tcp 1040 ------- 0 9.0 7.0 4 7
r 0.025813 0 9 ack 40 ------- 0 7.0 9.0 2 5
+ 0.025813 9 0 tcp 1040 ------- 0 9.0 7.0 5 8
+ 0.025813 9 0 tcp 1040 ------- 0 9.0 7.0 6 9
r 0.027813 9 0 tcp 1040 ------- 0 9.0 7.0 3 6
h 0.027813 0 8 tcp 1040 ------- 0 9.0 7.0 3 6
+ 0.028813 0 8 tcp 1040 ------- 0 9.0 7.0 3 6
- 0.028813 0 8 tcp 1040 ------- 0 9.0 7.0 3 6
- 0.029973 9 0 tcp 1040 ------- 0 9.0 7.0 5 8
r 0.03066 8 7 tcp 1040 ------- 0 9.0 7.0 3 6
h 0.03066 7 8 ack 40 ------- 0 7.0 9.0 3 10
+ 0.03166 7 8 ack 40 ------- 0 7.0 9.0 3 10
```

```
- 0.03166 7 8 ack 40 ------- 0 7.0 9.0 3 10
r 0.031973 9 0 tcp 1040 ------- 0 9.0 7.0 4 7
h 0.031973 0 8 tcp 1040 ------- 0 9.0 7.0 4 7
r 0.032715 8 0 ack 40 ------- 0 7.0 9.0 3 10
+ 0.032715 0 9 ack 40 ------- 0 7.0 9.0 3 10
- 0.032715 0 9 ack 40 ------- 0 7.0 9.0 3 10
+ 0.032973 0 8 tcp 1040 ------- 0 9.0 7.0 4 7
- 0.032973 0 8 tcp 1040 ------- 0 9.0 7.0 4 7
- 0.034133 9 0 tcp 1040 ------- 0 9.0 7.0 6 9
r 0.03482 8 7 tcp 1040 ------- 0 9.0 7.0 4 7
h 0.03482 7 8 ack 40 ------- 0 7.0 9.0 4 11
r 0.034875 0 9 ack 40 ------- 0 7.0 9.0 3 10
+ 0.034875 9 0 tcp 1040 ------- 0 9.0 7.0 7 12
+ 0.034875 9 0 tcp 1040 ------- 0 9.0 7.0 8 13
+ 0.03582 7 8 ack 40 ------- 0 7.0 9.0 4 11
- 0.03582 7 8 ack 40 ------- 0 7.0 9.0 4 11
r 0.036133 9 0 tcp 1040 ------- 0 9.0 7.0 5 8
h 0.036133 0 8 tcp 1040 ------- 0 9.0 7.0 5 8
r 0.036875 8 0 ack 40 ------- 0 7.0 9.0 4 11
+ 0.036875 0 9 ack 40 ------- 0 7.0 9.0 4 11
- 0.036875 0 9 ack 40 ------- 0 7.0 9.0 4 11
+ 0.037133 0 8 tcp 1040 ------- 0 9.0 7.0 5 8
- 0.037133 0 8 tcp 1040 ------- 0 9.0 7.0 5 8
- 0.038293 9 0 tcp 1040 ------- 0 9.0 7.0 7 12
r 0.03898 8 7 tcp 1040 ------- 0 9.0 7.0 5 8
h 0.03898 7 8 ack 40 ------- 0 7.0 9.0 5 14
r 0.039035 0 9 ack 40 ------- 0 7.0 9.0 4 11
+ 0.039035 9 0 tcp 1040 ------- 0 9.0 7.0 9 15
+ 0.039035 9 0 tcp 1040 ------- 0 9.0 7.0 10 16
+ 0.03998 7 8 ack 40 ------- 0 7.0 9.0 5 14
```

```
- 0.03998 7 8 ack 40 ------- 0 7.0 9.0 5 14
r 0.040293 9 0 tcp 1040 ------- 0 9.0 7.0 6 9
h 0.040293 0 8 tcp 1040 ------- 0 9.0 7.0 6 9
r 0.041035 8 0 ack 40 ------- 0 7.0 9.0 5 14
+ 0.041035 0 9 ack 40 ------- 0 7.0 9.0 5 14
- 0.041035 0 9 ack 40 ------- 0 7.0 9.0 5 14
+ 0.041293 0 8 tcp 1040 ------- 0 9.0 7.0 6 9
- 0.041293 0 8 tcp 1040 ------- 0 9.0 7.0 6 9
- 0.042453 9 0 tcp 1040 ------- 0 9.0 7.0 8 13
r 0.04314 8 7 tcp 1040 ------- 0 9.0 7.0 6 9
h 0.04314 7 8 ack 40 ------- 0 7.0 9.0 6 17
r 0.043195 0 9 ack 40 ------- 0 7.0 9.0 5 14
+ 0.043195 9 0 tcp 1040 ------- 0 9.0 7.0 11 18
+ 0.043195 9 0 tcp 1040 ------- 0 9.0 7.0 12 19
+ 0.04414 7 8 ack 40 ------- 0 7.0 9.0 6 17
- 0.04414 7 8 ack 40 ------- 0 7.0 9.0 6 17
r 0.044453 9 0 tcp 1040 ------- 0 9.0 7.0 7 12
h 0.044453 0 8 tcp 1040 ------- 0 9.0 7.0 7 12
r 0.045195 8 0 ack 40 ------- 0 7.0 9.0 6 17
+ 0.045195 0 9 ack 40 ------- 0 7.0 9.0 6 17
- 0.045195 0 9 ack 40 ------- 0 7.0 9.0 6 17
+ 0.045453 0 8 tcp 1040 ------- 0 9.0 7.0 7 12
- 0.045453 0 8 tcp 1040 ------- 0 9.0 7.0 7 12
- 0.046613 9 0 tcp 1040 ------- 0 9.0 7.0 9 15
r 0.0473 8 7 tcp 1040 ------- 0 9.0 7.0 7 12
h 0.0473 7 8 ack 40 ------- 0 7.0 9.0 7 20
r 0.047355 0 9 ack 40 ------- 0 7.0 9.0 6 17
```

The output of this tracefile clearly shows how different packets traversed the network and reached their destination correctly. Packets with names lik tcp, ack denote that they are tcp, acknowledgement packets respectively.

5.2.6 Testing with NAM Simulator

The simulator contains an efficient network animator called NAM. This animator can run the NAM format files generated by NS simulator.

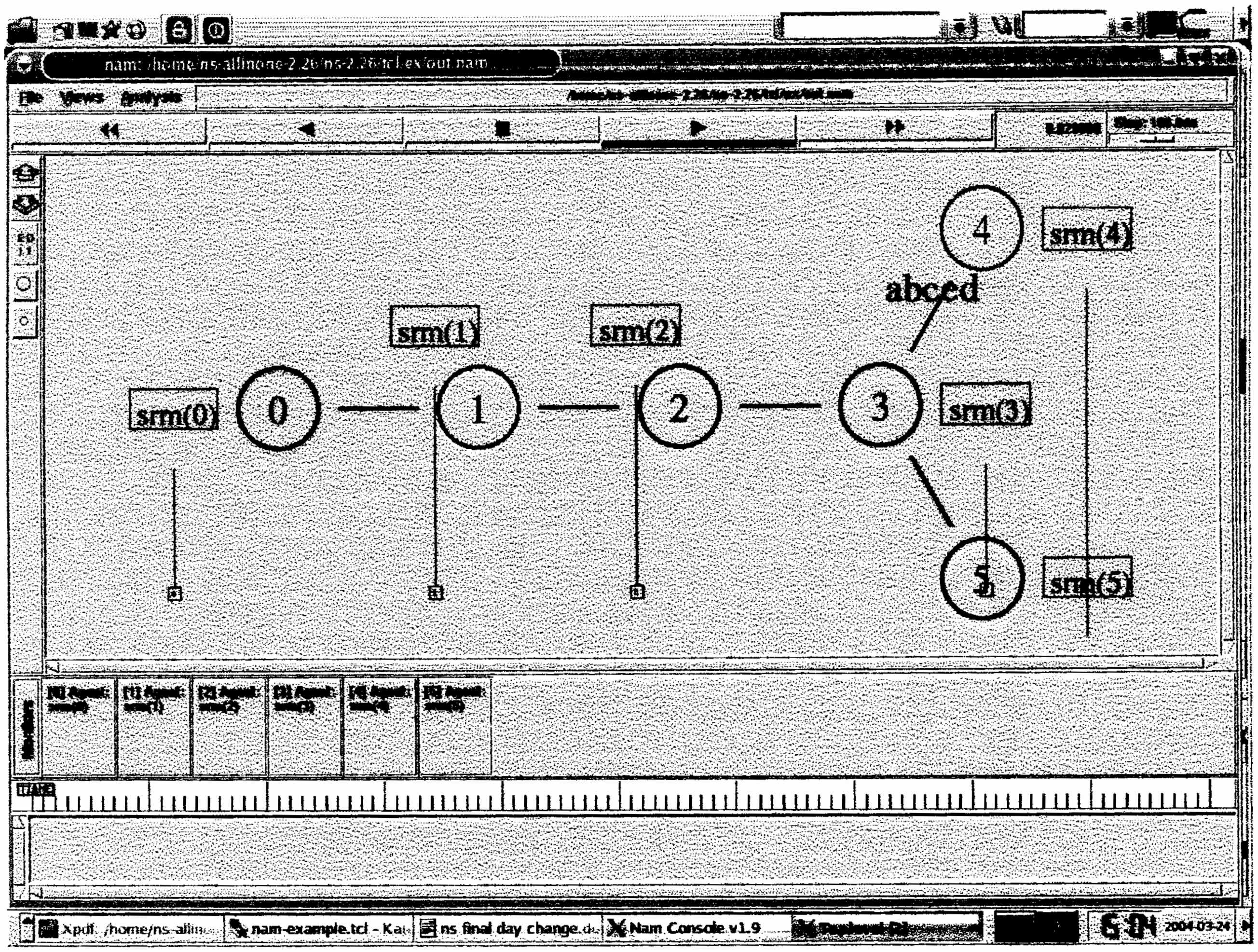


Fig 19: Testing with Nam Simulator

The above picture explains the architecture of NAM, the network animator of NS. The circles define the nodes, while the lines define the link between them. The buttons at the top control the flow of the animation. The play, pause and forward button are available.

5.2.7 Testing with Complex Topologies

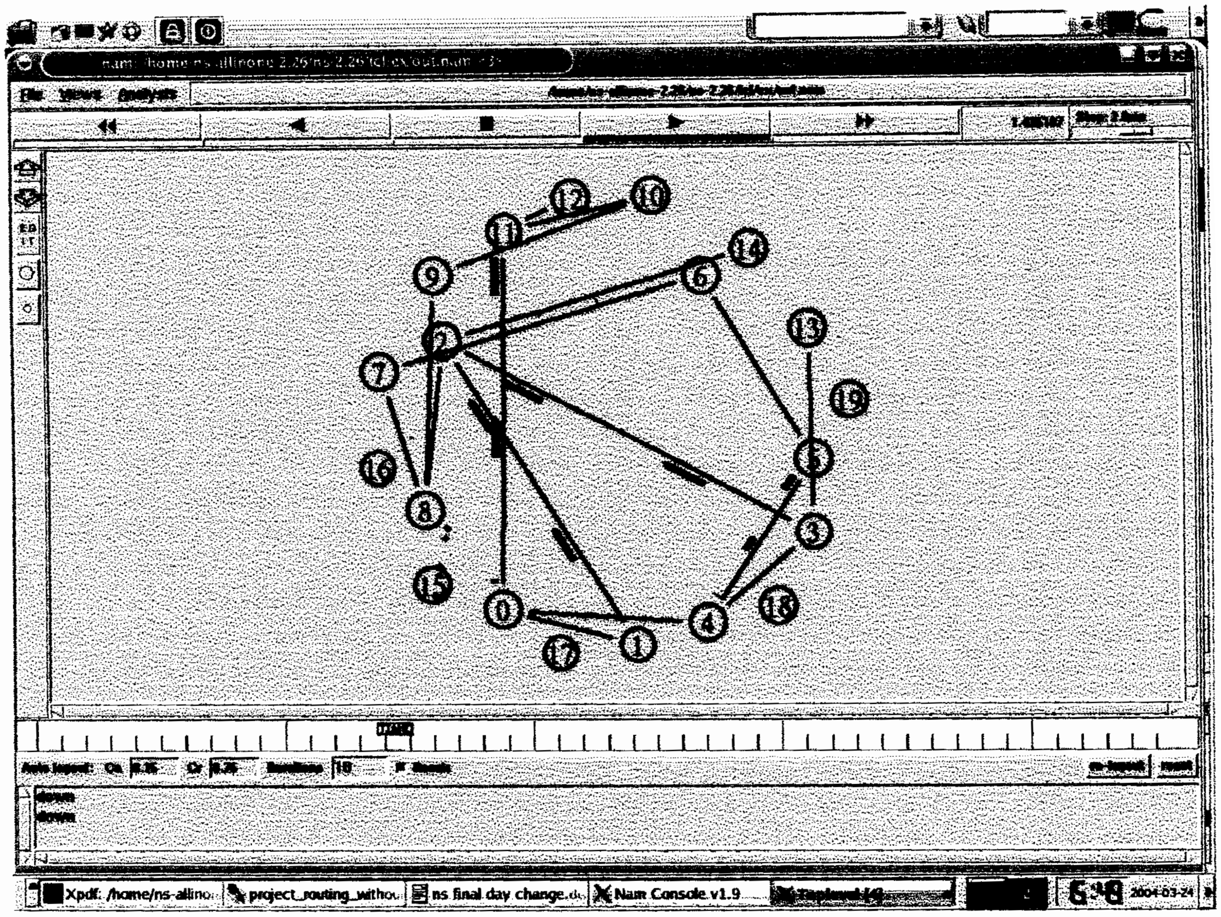


Fig 20: Testing with Complex Topologies

Making a complex topology with Q-protocol. Below is the file given.

```
#Create a simulator object
set ns [new Simulator]


$ns rtproto Q
#
#let down link invoke neighbor to neighbor updating
#
Agent/rtProto/Q set advertInterval 5
```

```
#Define different colors for data flows
$ns color 1 Blue
$ns color 2 Red

#Open the nam trace file
set nf [open out.nam w]
$ns namtrace-all $nf

$ns trace-all [open all.tr w]

#Define a 'finish' procedure
proc finish {} {
        global ns nf
        global downTimes
        $ns flush-trace
           #Close the trace file
        close $nf



           #Execute nam on the trace file
        exec nam out.nam &
        exit 0
}

set alllinks {{11 0 1MB 10ms } \
            {11 10 1MB 10ms } \
            {10 9 1MB 10ms } \
            {0 1 1MB 10ms } \
            {1 2 1MB 10ms } \
        {2 8 1MB 10ms } \
        {2 3 1MB 10ms } \
        {9 8 1MB 10ms } \
        {3 4 1MB 10ms } \
        {4 5 1MB 10ms } \
        {5 6 1MB 10ms } \
        {6 7 1MB 10ms } \
        {8 7 1MB 10ms } \
        {0 4 1MB 10ms } \
        {11 12 1MB 10ms} \
        {13 3 1MB 10ms}\
        {13 7 1MB 10ms}\
        {14 7 1MB 10ms}\
        {15 14 1MB 10ms }\
        {16 5 1MB 10ms }\
        {16 11 1MB 10ms }\
        {17 16 1MB 10ms } \
        {18 17 1MB 10ms} \
        {18 9 1MB 10ms} \
        {19 6 1MB 10ms}\
        }

for {set i 0} {$i < 20} {incr i} {
  set node($i) [$ns node]
}

foreach link $alllinks {
  set n1 [lindex $link 0]
  set n2 [lindex $link 1]
  set bw [lindex $link 2]
```

```
   set delay [lindex $link 3]
   #set dir [lindex $link 4]

   $ns duplex-link $node($n1) $node($n2) $bw $delay DropTail
   $ns queue-limit $node($n1) $node($n2) 3
   #$ns duplex-link-op  $node($n1) $node($n2) orient $dir
}

set n0 $node(0)
set n1 $node(4)
set n3 $node(8)

set dn1 $node(2)
set dn2 $node(8)


#Create a CBR agent and attach it to node n0
set cbr0 [new Agent/CBR]
$ns attach-agent $n0 $cbr0
$cbr0 set packetSize_ 1280
$cbr0 set interval_ 0.005
#$cbr0 set fid_ 1

#Create a CBR agent and attach it to node n1
set cbr1 [new Agent/CBR]
$ns attach-agent $n1 $cbr1
$cbr1 set packetSize_ 1280
$cbr1 set interval_ 0.005
#$cbr1 set fid_ 2
```

```
#Create a Null agent (a traffic sink) and attach it to node n3
#set null0 [new Agent/Null]
#$ns attach-agent $n3 $null0

 set sink_dst1 [new Agent/LossMonitor]

 set sink_dst2 [new Agent/LossMonitor]

 $ns attach-agent $n3 $sink_dst1

 $ns attach-agent $n3 $sink_dst2

 #Connect the traffic sources with the traffic sink
 $ns connect $cbr0 $sink_dst1
 $ns connect $cbr1 $sink_dst2

 #Schedule events for the CBR agents
 $ns at 0.5 "$cbr0 start"
 $ns at 0.5 "$cbr1 start"
 $ns at 4.5 "$cbr0 stop"
 $ns at 4.5 "$cbr1 stop"

 set downLn {{1 1.2} {1.4 1.6} {1.8 2.0} {2.2 2.4} {2.6 2.8} {3 3.2} {3.4 3.6} {3.8 4}}

 set cnt 0
 foreach du $downLn {
   if {$cnt == $downTimes } {
        break
   }
   set dtime [lindex $du 0]
```

```
    set utime [lindex $du 1]
    $ns rtmodel-at $dtime down $dn1 $dn2
    $ns rtmodel-at $utime up $dn1 $dn2

    incr cnt
}


#Call the finish procedure after 5 seconds of simulation time
$ns at 5.0 "finish"

#Run the simulation
$ns run
```

6. Conclusion and Future Work

6.1 Conclusion

In this thesis we presented an efficient and adaptive algorithm for network routing. The algorithm was implemented on NS simulator on Linux. The framework of Q-Routing is implemented and further research may be done over it. The algorithm is well suited for small-scaled as well as large-scale networks. Furthermore the architecture is modular and well defined making it easy to extend it.

The current implementation supports the following modules:

- Route Computation module
- Dynamic topology support module
- Distance Vector module
- Flooding support module

6.2 Future Work

The following tasks could be taken up as a part of future work, to take the work towards it logical completion.

6.2.1 Extensions of Q-Routing

Several extensions can now be implemented by researchers on the Q-Routing framework. These include but are not limited to: Q-Routing with confidence values, Q-Routing with dual reinforcement and using neural networks as function approximator.

6.2.2 Q-Routing on Wireless Ad-Hoc Networks

Very little work has been done in this area. Therefore anyone can take up this field and do research.

Appendix-A NS Simulator on Linux

Following steps were performed to run NS simulator on Linux.

A.1 Linux Installation

Redhat Linux 9.0 was used. The system installation went smoothly. Custom Installation was selected. Disk Druid window was loaded. The drive partitions were reformatted. Almost all the hardware was detected flawlessly by Linux and and after other settings the installation started. Redhat Linux 9.0 is now on 3 cds with a huge amount of software. The Tcl/Tk package is also installed which is needed by ns. The installation took almost 1:30 hours after which the system rebooted and the linux started.

A.2 NS Installation

First of all install the ns-allinone package. In ns-allinone-xxx\ns-2.26 folder the main ns package resides.

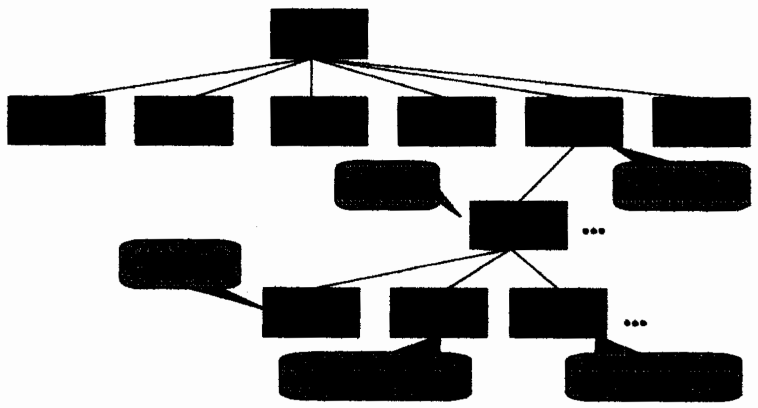
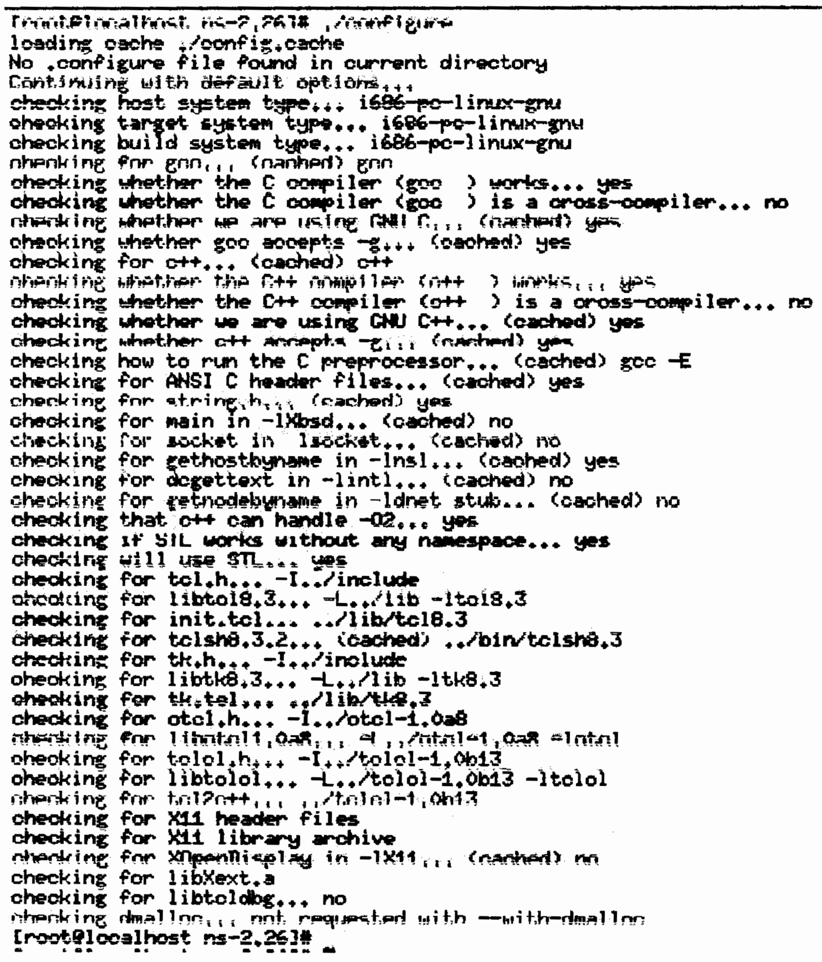


Fig 21: NS Directory Structure

Enter into the main folder of ns and type ./configure. This will result in the configuration of the NS simulator. Messages will be displayed and important information will be conveyed. Once this process is over without any errors, cursor reappears on the command line. Now type make. This will compile and install the ns-allinone package which contains Tcl, OTcl, Tclcl, xgraph etc. This is a lengthy process and requires a fast computer



```
[root@localhost ns-2.26]# ./configure
loading cache ./config.cache
No .configure file found in current directory
Continuing with default options...
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
checking build system type... i686-pc-linux-gnu
checking for gcc... (cached) gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... (cached) yes
checking whether gcc accepts -g... (cached) yes
checking for c++... (cached) c++
checking whether the C++ compiler (c++  ) works... yes
checking whether the C++ compiler (c++  ) is a cross-compiler... no
checking whether we are using GNU C++... (cached) yes
checking whether c++ accepts -g... (cached) yes
checking how to run the C preprocessor... (cached) gcc -E
checking for ANSI C header files... (cached) yes
checking for string.h... (cached) yes
checking for main in -lXbsd... (cached) no
checking for socket in  lsocket... (cached) no
checking for gethostbyname in -lnsl... (cached) yes
checking for dcgettext in -lintl... (cached) no
checking for getnodebyname in -ldnet stub... (cached) no
checking that c++ can handle -O2... yes
checking if STL works without any namespace... yes
checking will use STL... yes
checking for tcl.h... -I../include
checking for libtcl8.3... -L../lib -ltcl8.3
checking for init.tcl... ../lib/tcl8.3
checking for tclsh8.3.2... (cached) ../bin/tclsh8.3
checking for tk.h... -I../include
checking for libtk8.3... -L../lib -ltk8.3
checking for tk.tcl... ../lib/tk8.3
checking for otcl.h... -I../otcl-1.0a8
checking for libotcl1.0a8... -L../otcl-1.0a8 -lotcl
checking for tclcl.h... -I../tclcl-1.0b13
checking for libtclcl... -L../tclcl-1.0b13 -ltclcl
checking for tcl2c++... ../tclcl-1.0b13
checking for X11 header files
checking for X11 library archive
checking for XOpenDisplay in -lX11... (cached) no
checking for libXext.a
checking for libtcldbg... no
checking dmalloc... not requested with --with-dmalloc
[root@localhost ns-2.26]# _
```

A.3 NS configuration and protocol addition

After installation of NS simulator, we need to configure it. For many components of the NS simulator to work properly, we need to set the library path of these components at appropriate places i.e., xgraph, tclcl etc. We have to set LD_LIBRARY_PATH. Also we need to set the path of required NS folders in the bash or csh login file. Once the path has been set and libraries have been linked, we can start using the NS simulator.

The ns simulator runs by the help of two languages c++ and tcl. So for running any new protocol we need to add c++ as well as tcl code to the simulator.

For the c++ code we have to write our c++ code in two files. (file.cc and file.h) Suppose they are pong.cc and pong.h Place these files in the apps folder of ns which is ns-allinone-2.26/ns-2.26/apps. Note that these files can be placed in other directories also. Only the makefile has to be modified to include the path and filenames of these files. Now edit the file ns-allinone-2.26/ns-2.26/common /packet.h to add new packet entry. In this case it is

```
PT_PONG

name_[PT_PONG]=pong
```

For the tcl code we need to write our code in a tcl file and modify some tcl files. First of all modify the file ns-allinone-2.26/ns-2.26/tcl/lib/ns-default.tcl to place our default values there also. In this case they are

```
Agent/Pong set packetSize_ 64
Agent/Pong set off_pong_ 0
```

These are the default values in tcl for the new agent Pong which we have written in pong.cc and pong.h.

Now edit the file ns-allinone-2.26/ns-2.26/tcl/lib/ns-packet.tcl to enter the new packet type which we have defined in pong.cc and pong.h. They are

```
foreach prot {
MIP
     Pong
}
```

After the above two steps execute ns-allinone-2.26/ns-2.26/make depend. This will resolve the dependencies. Then write ns-allinone-2.26/ns-2.26/make. This will recompile the ns package and if completed successfully our written c++ code of pong.cc and pong.h would be linked into the ns.

Once the above c++ code is correctly compiled and linked into the main ns package and the tcl modification in files is also done and a pong.tcl file for executing our new protocol is made then we can go to the following folder ns-allinone-2.26/ns-2.26 /apps/ns pong.tcl and the simulation will start.

Appendix-B Glossary of Terms

This appendix contains terminology that is related to computer networks

Bandwidth: Total link capacity of a link to carry information (typically bits)

Channel: The physical medium is divided into logical channel, allowing possibly shared uses of the medium. Channels may be made available by subdividing the medium into distinct time slots, distinct spectral bands, or de-correlated coding sequences.

Flooding: The process of delivering data or control messages to every node within the data network.

Host: Any node that is not a router.

Link: A communication facility or medium over which nodes can communicate at the link layer.

Loop free: A path taken by a packet never transits the same intermediate node twice before arrival at the destination.

Next hop: A neighbour, which has been designed to forward packets along the way to a particular destination.

Neighbor: A node that is within transmitter range from another node on the same channel.

Node: A device that implements IP

Node ID: Unique identifier that identifies a particular node

Neighbour-node: A node that is one hop away.

Router: A node that forwards IP packets not explicitly addressed to itself. In case of ad hoc networks, all nodes are at least unicast routers.

Routing table: The table where the routing protocols keep routing information for various destinations. This information can include the next hop and the number of hops to the destination.

Scalability: A protocol is scalable if it is applicable to large as well as small populations.

Throughput: The amount of data from a source to a destination processed by the protocol for which throughput is to be measured for instance, IP, TCP, or the MAC protocol.

BIBLIOGRAPHY AND REFERENCES

[1] Tanenbaum, A. S., Computer Networks. 3th Ed. Prentice Hall PTR, Upper Saddle River, New Jersey, U.S.A, 1996.

[2] William Stalling, Modern Computer Networks :, Third Edition, Prentice Hall, USA, 1997.

[3] R. Bellman. On a routing problem. Quarterly of Applied Mathematics, 16(1):87--90, 1958.

[4] Kaelbling, Littman, Andrew W. Moore. Reinforcement Learning: A Survey. Journal of Artificial Intelligence Research 4 (1996) 237-285.

[5] G. Tesauro. Practical issues in temporal difference learning. Machine Learning, 8(3/4), May 1992.

[6] C. Watkins. Learning from Delayed Rewards. PhD thesis, King's College, Cambridge, 1989.

[7] Packet Routing in Dynamically Changing Networks: Reinforcement Learning Approach, Justin A. Boyan, Michael L. Littman.

[8] D Choi, S.P.M, and Yeung, D.-Y. Predictive Q-Routing: A memory-based reinforcement learning approach to adaptive traffic control. 1996.

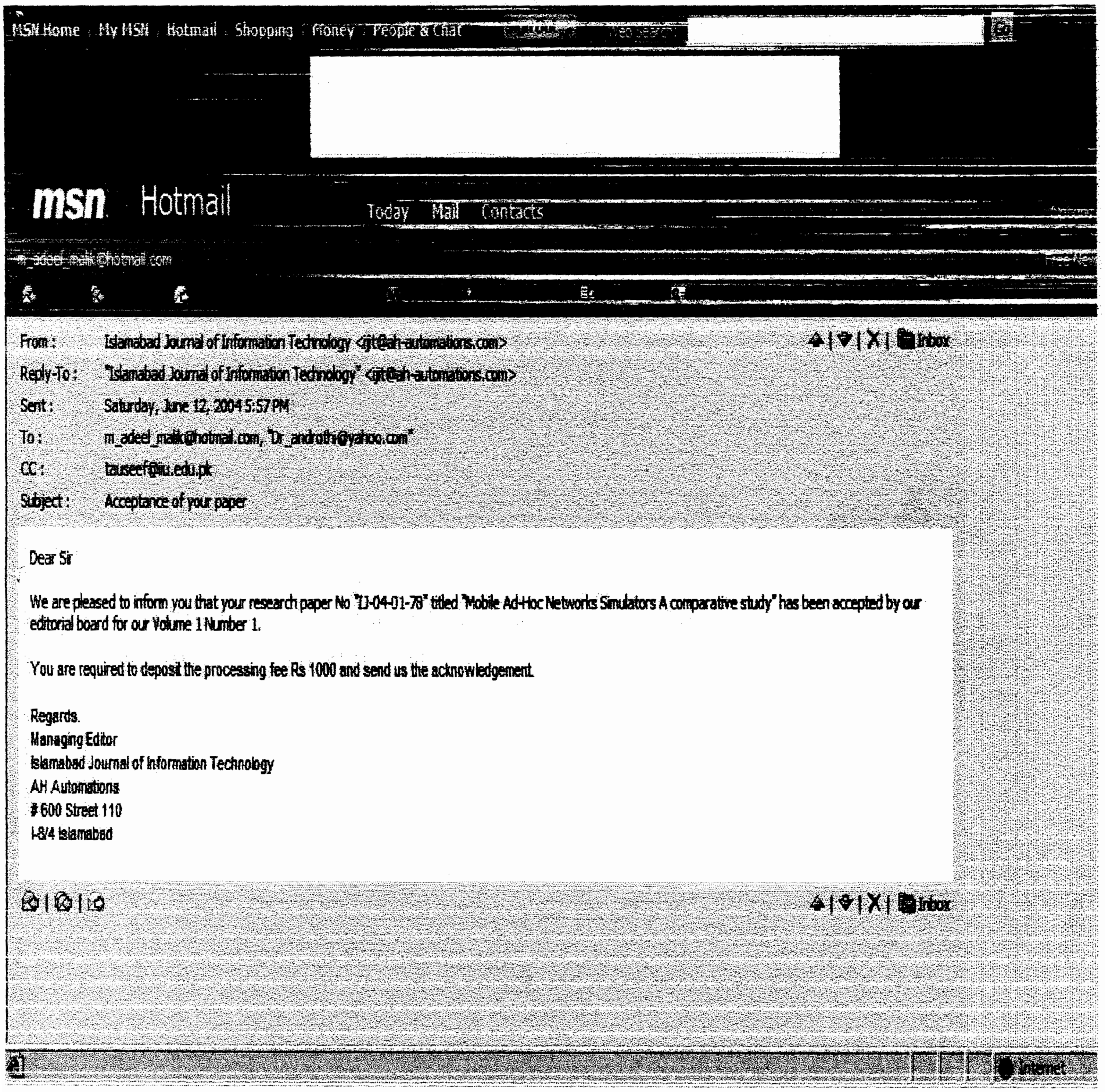
[9] Confidence based Dual Reinforcement Q-Routing: an On-line Adaptive Network Routing Algorithm, Masters Thesis of Shailesh Kumar, University of Texas, Austin.

[10] www.isi.edu/nsnam

[11] http://www.isi.edu/nsnam/ns/ns-documentation

[12] http://www.lam-mpi.org/

RESEARCH PAPER


MSN Home | My MSN | Hotmail | Shopping | Money | People & Chat
msn Hotmail
Today | Mail | Contacts
m_adeel_malik@hotmail.com
Inbox
Internet

From : Islamabad Journal of Information Technology <ijit@ah-automations.com>
Reply-To : "Islamabad Journal of Information Technology" <ijit@ah-automations.com>
Sent : Saturday, June 12, 2004 5:57 PM
To : m_adeel_malik@hotmail.com, "Dr_androthi@yahoo.com"
CC : tauseef@iiu.edu.pk
Subject : Acceptance of your paper

Dear Sir

We are pleased to inform you that your research paper No "IJ-04-01-78" titled "Mobile Ad-Hoc Networks Simulators A comparative study" has been accepted by our editorial board for our Volume 1 Number 1.

You are required to deposit the processing fee Rs 1000 and send us the acknowledgement.

Regards.
Managing Editor
Islamabad Journal of Information Technology
AH Automations
600 Street 110
I-8/4 Islamabad

Mobile Ad-Hoc Networks Simulators: A Comparative Study

[1]Dr. Tauseef Ur Rahman, [2]Muhammad Adeel
[1]Department of Telecom and Computer Engineering, International Islamic University Islamabad, Pakistan
[2]Department of Computer Science, International Islamic University Islamabad, Pakistan
{[1]tauseef@iiu.edu.pk, [2]m_adeel_malik@hotmail.com}

Abstract: A mobile Ad-hoc network is the cooperative engagement of a collection of mobile nodes without the required intervention of any centralized access point or existing infrastructure. Research in the field of mobile Ad-hoc networks requires the use of network simulators. In this paper we compare two popular network simulators; OMNet and NS. Comparison study shows that in most cases NS simulator is more efficient and robust for the purpose of simulation of mobile Ad-Hoc networks.

Keywords: Mobile Ad-Hoc Networks, OMNet++, NS.

1. INTRODUCTION

For experiments involving network research, a simulator can be used to freely place and modify network components as per required [3]. Helpful debugging and execution control proves to be very helpful.

NS and OMNet are two network simulators which are getting increasingly used for network research. Both are open source and thus can be freely modified. Features of OMNet++ and NS are described in section II and then section III. Then a detailed comparison is done in section IV. Finally conclusion is presented in section V.

2. OMNet++

OMNet++ is a discrete event simulator using C++ as the simulator language. The name stands for Objective Modular Network Testbed in C++. The GUI is written in Tcl/Tk. This is a portable simulator and works on windows and several unix flavours.[1] The simulator can be used for

- Testing new algorithms
- Protocol modeling
- Multiprocessors and distributed hw systems

In general wherever we use discrete event approach we can use OMNet [1]. OMNet introduces the concept of models which may represent networks of any type and size. A model can contain modules which may contain further sub modules. The logical structure of the network can hence be fully represented.

For communication purposes the modules exchange information with each other through message passing. They can do it directly or through gates and connections. Gates are the input and output interfaces of modules. Messages are sent out through output gates and arrive through input gates. Links or connections are used between modules to connect them with each other (as shown in Figure).

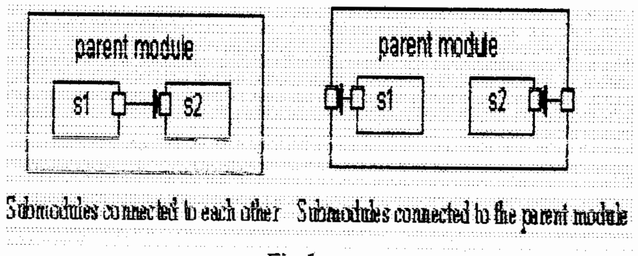

parent module
s1
s2
parent module
s1
s2
Submodules connected to each other
Submodules connected to the parent module

Fig 1

Connections going from simple to simple modules are also called routes. These connections may be assigned parameters like propagation delay, bit error rate and data rate.

OMNet modules have parameters which serve the following purpose

- Creating flexible topologies
- Customizing module behavior
- Module communication

User defined algorithms are implemented in simple OMNet modules. Simple modules are implemented as co-routines so they appear to execute in parallel. User should have some knowledge of C/C++ to do this.[1]

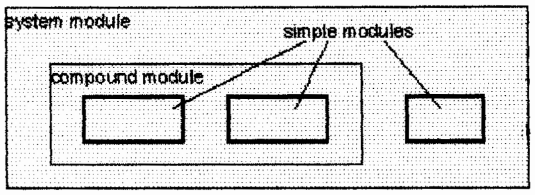


Fig 2

Modules communicate by exchanging messages. Messages can be representing real life network packets/frames, jobs or customers in a mobile network or other mobile entities.

OMNet uses NED language for network topology description [1]. This language is specially designed for network description. It supports modular description of a network. The network descriptions made in this language are reusable which means one NED network description can be used in another network.

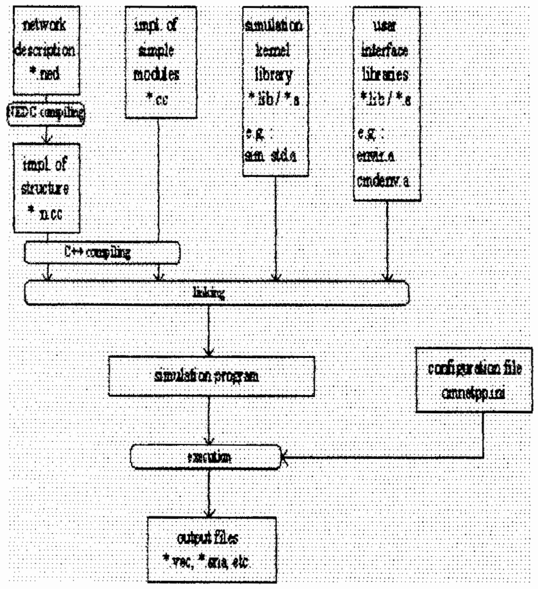

network description *.ned
NEDC compiling
impl. of structure *.n.cc
impl. of simple modules *.cc
simulation kernel library *.lib / *.a e.g.: sim_std.a
user interface libraries *.lib / *.a e.g.: envir.a cmdenv.a
C++ compiling
linking
simulation program
configuration file omnetpp.ini
execution
output files *.vec, *.sca, etc.

Fig 3

OMNet provides cross platform compatibility. This allows the simulation to be run on Windows, Linux and many other flavors of UNIX. Microsoft Visual C++ integration is also provided on Windows platform. MSVC IDE may be used for debugging of the simulation. OMNet++ utilizes the macros of MSVC to integrate OMNet++ with Visual C++.

3. NS

NS is an object oriented simulator. Developed at UC Berkeley it simulates different networks, implements network protocols such as TCP,UDP, traffic source behavior such as FTP, Telnet, CBR, routing algorithms such as Dijkstra and more. It also implements multicasting and some MAC layer protocols. It uses a unique split programming concept for providing its functionality. The two languages which ns uses are C++ and otcl (an extension of the popular tcl scripting language). Correspondingly there are two class hierarchies in ns.[2]

- Compiled Hierarchy
- Interpreted Hierarchy

Both these hierarchies are closely related with each other. OTcl is the object oriented version of Tcl language and it has the same relation- ship with Tcl as C++ has with c [7].

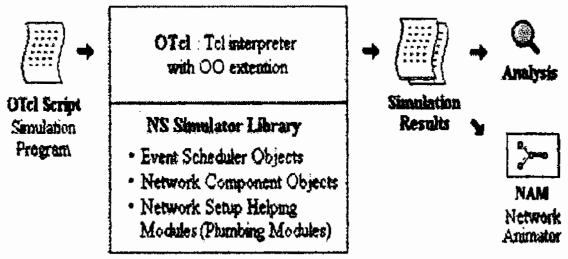

OTcl : Tcl interpreter with OO extention
NS Simulator Library
• Event Scheduler Objects
• Network Component Objects
• Network Setup Helping Modules (Plumbing Modules)
OTcl Script Simulation Program
Simulation Results
Analysis
NAM Network Animator

Fig 4

As shown in figure NS contains an object oriented Tcl (OTcl) interpreter that has a simulation event scheduler and network component object and network setup libraries.

NS is written in two languages C++ and OTcl. This is mainly done for efficiency reasons. The event scheduler and the basic network objects are written and compiled in C++. Through a unique binding mechanism the compiled objects of C++ are made available to the OTcl interpreter [2].

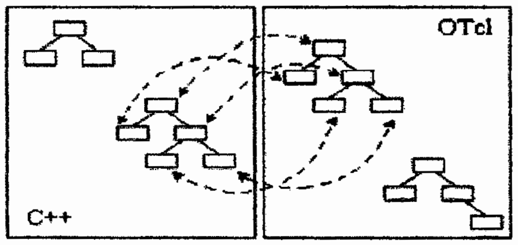


Fig 5

We can say that NS is a Object Oriented Tcl interpreter with network simulator libraries. NS has a well thought architecture [2]. This was particularly demonstrated when the wireless/satellite support was added to NS seamlessly.

NS contains a graphical simulation display tool called NAM. NAM is used extensively to graphically visualize different simulations. It is a very capable tool and can present information such as throughput and number of packet drops at each link, although accurate simulation analysis cannot be done with that data.

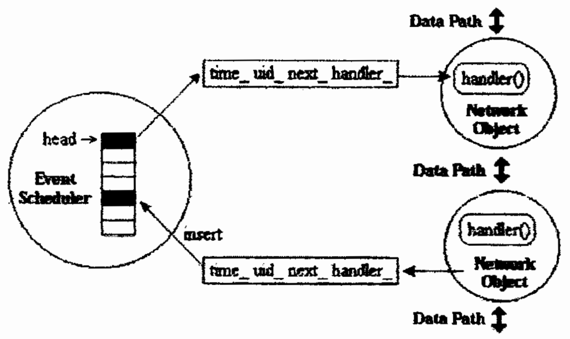


Fig 6

NS contains a discrete event scheduler. The main users of an event scheduler are network components that simulate packet-handling delay or that need timers. Above figure shows the NS event scheduler in action. Packets are send from one network object to another by using send (Packet* p) { target_->recv(p)}; for the sender and recv (Packet* , Handler* h = 0) for the receiver [2].

One very significant addition to the NS simulator is that of a real-time scheduler allowing NS to integrate in a real life LAN and introduce packets into it. This opens up some very exciting possibilities.

A simple NS simulation may consist of 2 nodes and a duplex link between them [2]. A time interval may be set using a scheduler after which the nodes may start communication with each other. A stop time may also be specified. This whole simulation can be written in about 10 lines of OTCL script.

4. COMPARISON

A comparison between features of NS and OMNet was performed. Different criteria like detail level, model availability, parallel execution etc was selected for comparing the two simulators with each other.

Detail Level

OMNet++ provides sufficient detail for the user to implement new building blocks in his simulation. User can easily write new classes to extend the functionality of OMNet++[1].

NS has a relatively complex method to extend the simulator. A module has to be written in C++ and then it's equivalent class should be made available in the interpreted hierarchy and both should be linked with each other.

Model Availability

Model availability is a important criteria. It necessarily means what protocol models are available to the user to use in his custom simulation. OMNet++ was found to have relatively less collection of network models while NS contains an extensive collection of network models.

Parallel Execution

OMNet++ modules use message passing as the primary communication mechanism between its modules. Both MPI and PVM are supported. OMNet++ uses a conservative approach to provide parallel discrete event simulation (PDES). This approach is implemented by using Null Message Algorithm (NMA) [5].

This approach has essentially the drawback of reverting back to sequential simulation if parallelism in the model is not fully exploited or enough null messages are not sent to participating nodes. (Parallel simulation support is currently being redesigned in OMNet++).

Debugging Support

OMNet++ supports command line as well as off-line and client-server style debugging. It does so by linking the GUI library with the debugging/tracing capability into the simulation executable. This powerful approach allows full control over the execution.

NS provides debugging support by generating trace files for text and graphical output [2][7]. Runtime debugging is complex and time consuming because of debugging issues involving both Tcl and C++. Although a debugger for debugging Tcl is available, it requires to be compiled and linked with ns, the simulation script may be modified to add debug enable lines, and the whole NS package may be recompiled. While running the simulation on Linux the switching between the GNU debugger and Tcl debugger was quite complex and time consuming.

Topology Description and Generation

OMNet++ follows a unique way for description and generation of network topology [1]. It uses a custom built network description language called (NED) for this purpose. Both command line as well as gui interface is provided. This allows the user to use one NED file with network description in another project. Internally the NED files are translated into C++ code, then compiled by the C++ compiler and then linked into the simulation executable.

NS uses tcl for topology generation which provides for very easy topology generation. Hierarchies like those of OMNet are not forced. More control is provided to the user. Creating complex hierarchies in NS was therefore found to be more time consuming than in OMNet++ [2].

Programming Model

OMNet allows programming in C++ with message passing and co-routine execution. Integration with MSVC is also provided [1].

NS provides an easy programming model with OTcl and C++. If the built-in C++ objects are used as is in OTcl then a simple mobile ad-hoc network with AODV can be formed in about 20 lines of code. Something unthinkable when using OMNet for the same purpose [2][7].

Performance

A sample simulation of OMNet was compared with identical simulations developed in C and PARSEC languages. The results were very encouraging as OMNet was only 1.3 times slower than the simulation developed in plain C. Similar results were achieved in the second case [1].

NS was found to be relatively slow due to the binding and linking overhead between the two languages OTcl and C++ [2][7].

5. CONCLUSION

In this paper two popular network simulators OMNet++ and NS-2 were compared with each other. It was shown that NS is easy to use when a simulation is built using the built-in C++ objects in the OTcl interpreter but it is difficult to extend it due to the use of two languages OTcl and C++. On the other hand OMNet++ was found to be relatively easy to extend as the whole class hierarchy was in C++.

6. REFERENCES

[1] András Varga, OMNet ++ User Manual
www.omnet.org

[2] NS manual and examples
www.isi.edu/nsnam/ns/ns-documentation

[3] Tanenbaum, A. S., Computer Networks.
3th Ed. Prentice Hall PTR, Upper Saddle River, New Jersey

[4] Charles E. Perkins and E.M.Royer, Ad- Hoc On Demand Distance Vector Routing

[5] Y. Ahmet Sekercioglu, Andras Varga and Gregory K. Egan, Parallel Simulation Made Easy with OMNet ++.

[7] NS documentation and source available from
www.isi.edu/nsnam