The University of Kansas

Information and Telecommunication Technology Center

Technical Report

Emulation of RDRN on an ATM-Testbed and a Comparative Evaluation of IP vs ATM

Syed Fazal Ahmad, Gary Minden and
Joseph Evans

ITTC-FY2000-TR-13380-08

September 1999

Project Sponsor:
Information Technology Office
of the
Defense Advanced Research Projects Agency

Abstract

Multi-hop mobile wireless network (e.g. Rapidly Deployable Radio Networks, RDRN) is an ideal technology to establish instant communication infrastructure for civilian as well as military purpose. Before, such a network is actually deployed in the "real world", it is imperative to understand the behavior of the system in a large-scale network under different scenarios. Hence, one of the purposes of this research was to design and provide a controlled, repeatable and realistic environment over which the emulation of the RDRN software could be performed. The second objective after the design of the emulation environment was to do a comparative evaluation of IP vs ATM connectivity between the nodes and to measure different performance metrics (throughput, time for the network to converge and fraction of the emulation time for which connectivity could be maintained).

Table of Contents:

Index of Figures:

Index of Tables:

The point of philosophy is to start with something so simple as to seem not worth stating, and to end with something so paradoxical that no one will believe it.

Bertrand Russell

Chapter 1

Motivation & Background

In this chapter the motivation behind, the aims & the objectives of this work has been discussed. Moreover, there is a brief overview of the Rapidly Deployable Radio Network (RDRN) and its various components.

1.1 Motivation & Purpose

Conducting large-scale tests of mobile-networks, like the Rapidly Deployable Radio Networks (RDRN), is an expensive job, both financially as well as in terms of the effort required. But it is imperative to understand and analyze the behavior of the system on a large-scale under different conditions before deployment can be considered.

There are three possible ways of doing such large-scale tests. They are:

- Use a network simulator and implement the desired system in it.
- Field Tests
- Develop an emulation environment over a testbed where the existing software modules can be used with minimal changes.

The field tests would provide the most accurate indicator of the performance, the robustness and the scalability of the system. However, the field tests were not a feasible option because of the large set of radios required and the logistics. Moreover, it would be difficult to repeat the tests under the same conditions. Hence, we chose to design an emulation environment for conducting large-scale tests for the RDRN for it would provide a more accurate measure of the concerned metrics than simulating the system on a network simulator.

Hence, the first objective of this work was to design and implement a true, a repeatable, a controlled and a scalable emulation environment for the Rapidly Deployable Radio Network (RDRN).

The second objective after designing and implementing the emulation environment was to run different scenarios and to measure performance metrics like throughput, time taken for the network to converge and the fraction of the emulation time during which connectivity could be maintained. The throughput was measured for two cases: a) IP connectivity between the nodes, and b) ATM connectivity between the nodes. The purpose of this was to do an initial comparative evaluation of IP vs ATM for a highly dynamic environment like RDRN. These metrics would also provide an insight into the scalability and the performance of the software modules (network controller).

1.2 Rapidly Deployable Radio Network (RDRN)

1.2.1 Introduction[1]:

The objective of the Rapidly Deployable Radio Network (RDRN) is to investigate wireless ATM technology for high performance, adaptive, rapidly self-configuring mobile radio networks in the framework of the Global Mobile Information Systems (GloMo) programs initiated by DARPA.

The RDRN architecture represents a different approach to ATM-based wired/wireless network architectures. The overall system features the combination of two wireless network topologies:

i. A low-speed packet radio technology to enable reliable out-of-band control for the rapid deployment of point-to-point wireless ATM links, and
ii. A high-speed beamforming radio technology over which point-to-point wireless ATM links is established.

As a result the RDRN design envisions the deployment of highly dynamic topologies which require end-to-end ATM networking support, seamless interoperability with legacy IP-based networks, and multi-hop operation over wireless nodes that can be as far as 10 km in distance.

There have been several projects that have attempted architectures for wireless ATM such as RedNet[2], BAHAMA[3], WATMNet[4], etc. but the features that distinguish the RDRN system from the others are:

i. Architecture composed of two overlaid radio networks.
ii. Network configuration, control and management algorithms based on location management distributed across the packet radio network.

iii. Phased array antenna with digital beamforming and software radio
iv. Mobility-aware software-based ATM switch on edge nodes.
v. Adaptive wireless communication protocols.

1.2.2 Overview:

The main objective of the RDRN architecture is to use an adaptive point-to-point topology to gain the advantages of the wireless networks. Figure 1 shows a high-level view of the RDRN system, which is made up of two types of nodes:

i. Mobile End Points (MEPs) to provide wireless ATM access to end-users, and
ii. Mobile Access Points (MAPs) serving as radio access point to enable switching and connectivity among MEPs users and the ATM-LAN.

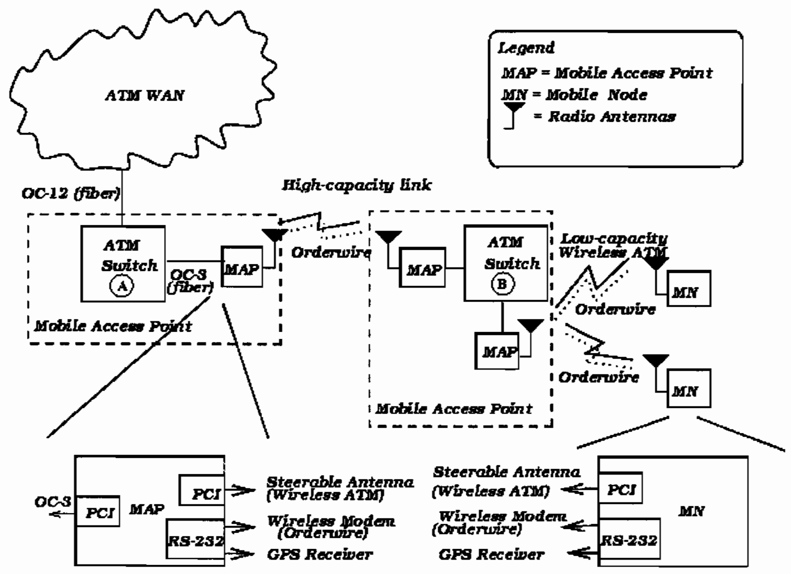


Figure 1: High-Level View of RDRN[1]

The architecture is composed of two overlaid networks:

i. A low bandwidth, low power, omni-directional network, called the orderwire network, intended for local dissemination, topology configuration, and link setup management among RDRN nodes.

ii. A high bandwidth, multi-directional network, called the WATM(Wireless ATM) network, that includes
    - 1 Mbps point-to-point connectivity between the MEP and a MAP.
    - 2 Mbps point-to-point connectivity among MAPs.

The radio networks have been tested over a distance as far as 7 kilometers. The orderwire network provides a coarse-grain control mechanism for managing links to be setup over the WATM network while the WATM network provides a fine-grain control mechanism for controlling the resources within the established links in addition to transporting the user data. As described above, the RDRN system consists of two wireless network topologies. The reasons for this are:

i. Because of its low-speed and low frequency, the orderwire offers a high level of reliability which is key to successful establishment and continuos adaptation of the point-to-point links over the not-so-reliable WATM network.
ii. This also leads to a simplified design of the overall system.

1.2.3 RDRN System Implementation:

1.2.3.1 RDRN Hardware:

Each node in the RDRN system (i.e. MAP and MEP) can be best described as a transportable unit. This unit is equipped with a laptop computer, a Global Positioning System (GPS) receiver, a 9,600 bps packet radio transceiver, and a custom-designed phased-array steerable antenna system The antenna system features an omni-directional receiver, a single-directional (multiple-directional) beamforming transmitter for the MEP (MAP), and a full-duplex connection to the wireless ATM adapter installed on the laptop. When initially deployed, each RDRN node retrieves its location information from the GPS receiver. A MAP is capable of forming multiple digitally formed beams in the directions of other MAPs or towards other MEPs in the vicinity. Multiple digital beams formed by a single transmitter are all of the same frequency to allow for spatial frequency reuse. The data rate available on the antenna is 2 Mbps between the MAPs and 1 Mbps between the MAPs & MEPs. Figure 2 shows an overview of the link-level mechanism utilized in the RDRN system.

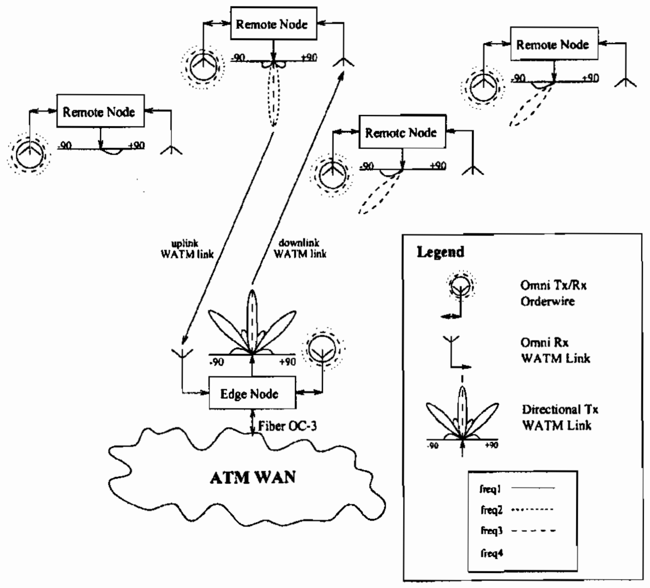


Figure 2: Link Level View of the RDRN [1]

1.2.3.2 RDRN Software:

Linux is the operating environment of the RDRN system running on both the MAPs and the MEPs. The RDRN software can be divided into three major components:

- Orderwire modules
- WATM Module
- Routing module

1.2.3.2.1 The Orderwire Network:

The wireless topology is setup by initially having the MAPs broadcast their position over the orderwire network and listen for location broadcasts from the other RDRN nodes. Similarly, MEPs broadcast their position over the orderwire system. A location-based distributed network configuration algorithm is executed to establish link-level connectivity among the MAPs and sets of MEPs or among the adjacent MAPs. The algorithm controls the assignment of node-to-node link connection, the assignment of beam to users and the handoff of users from one MAP to another. As RDRN nodes move, position updates from the GPS receivers are received and the design goal is to use this to steer the beams in the correct direction.

The topology algorithm works differently on the MAPs and the MEPs. On the MAPs, the algorithm calculates the distance between itself and all the other nodes it has heard from. Since the range of the orderwire is different from that of the high-speed link, the topology algorithm checks whether the calculated distances are within the range for high-speed connectivity or not. If the calculated distances are within the range for high-speed connectivity, it tries to establish point-point links with the nearest four, each being on a different beam. On the MEPs, the algorithm calculates the distance between itself and all the MAPs it has heard from. If the above-calculated distances are within the range for high-speed connectivity, it tries to establish the point-to-point link with the *nearest* MAP.

Once the RDRN network topology is initialized at the WATM link-level using the orderwire, the RDRN nodes start their configuration at the ATM level over the WATM network. Local WATM link configuration strictly follows the orders the orderwire to setup, adapt, and tear-down WATM point-to-point links. Point-to-point WATM links are established at the physical level by close cooperation between the orderwire and the beamforming antenna system and that communicates through the WATM adapter.

1.2.3.2.2 The WATM network:

The wireless access in the RDRN network is not restricted to the last hop. Figure 3 describes the protocol architecture for the WATM system. The WATM modules are a mix of user-level programs and kernel drivers embedded into the Linux-ATM software. Linux-ATM is used to provide native-mode ATM as well as TCP/IP over ATM support to run applications and user-level signaling programs. The WATM protocol stack on the MEP looks like any other ATM device driver to Linux-ATM. Similarly, on the MAP, multiple WATM protocol stacks (and a single ATM protocol stack) looks like any other ATM device driver to Linux-ATM. However on the MAPs, ATM virtual circuit (VC) packets are routed through the Micro-Switch if configured to use AAL0 (null) encapsulation; else, they are treated as AAL5 packets and processed accordingly by the Linux-ATM architecture.

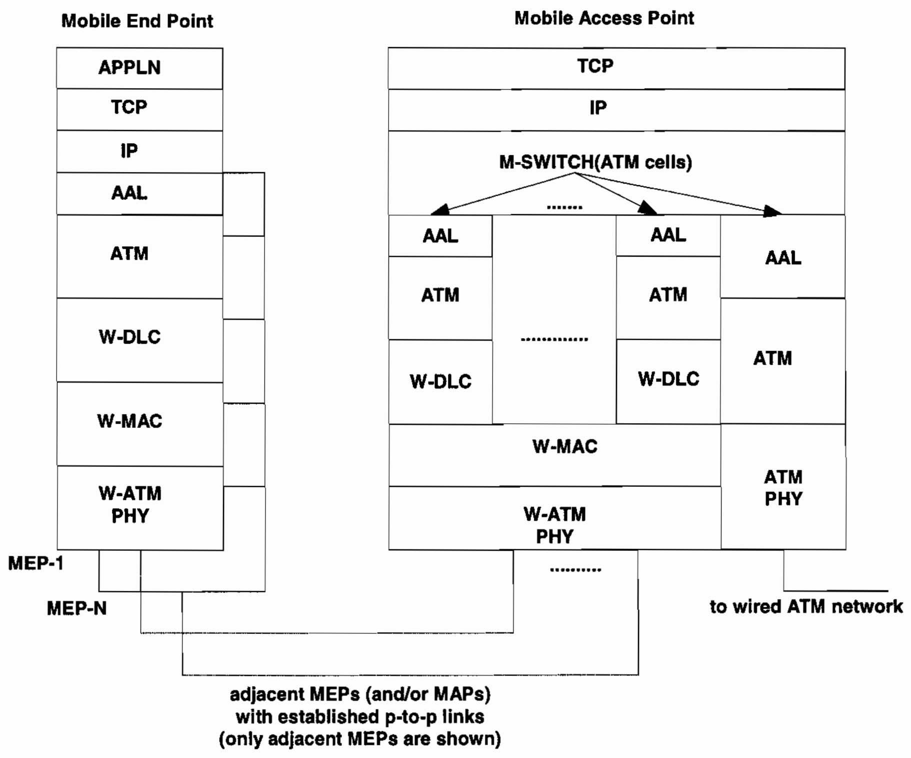


Figure 3: WATM Protocol Architecture[1]

Air packets transmitted over the WATM network are encapsulated using the WATM frame format shown in Figure 4. The WATM stack is composed of several layers: the AAL (ATM adaptation layer), the ATM layer (segmentation and re-assembly), the W-DLC layer (data link control enhanced for wireless), a W-MAC layer (medium access control for wireless) and a W-PHY layer (physical for wireless). The AAL layer provides an interface between the WATM stack and the Linux-ATM architecture. The ATM layer performs ATM segmentation and re-assembly functions for the AAL5 and AAL0 (null) encapsulation types. The W-DLC layer performs link control operations to transmit a set of ATM cells over a point-to-point WATM link. The number of ATM cells included on a set is negotiated between peer W-DLC layers and adjusted depending upon the conditions of the particular WATM link. The W-MAC layer in the RDRN system is somewhat simplified since the WATM network already assumes point-to-point WATM link connectivity. The W-MAC header contains a link-level address, frame type (timesensitive data, loss-sensitive data, or control), encoding scheme (not implemented yet) and the number of ATM cells (for data type frame) or control type (for control type frames). The W-PHY layer appends a header to the WATM frame for channel equalization and timing at the physical level on the receiver.

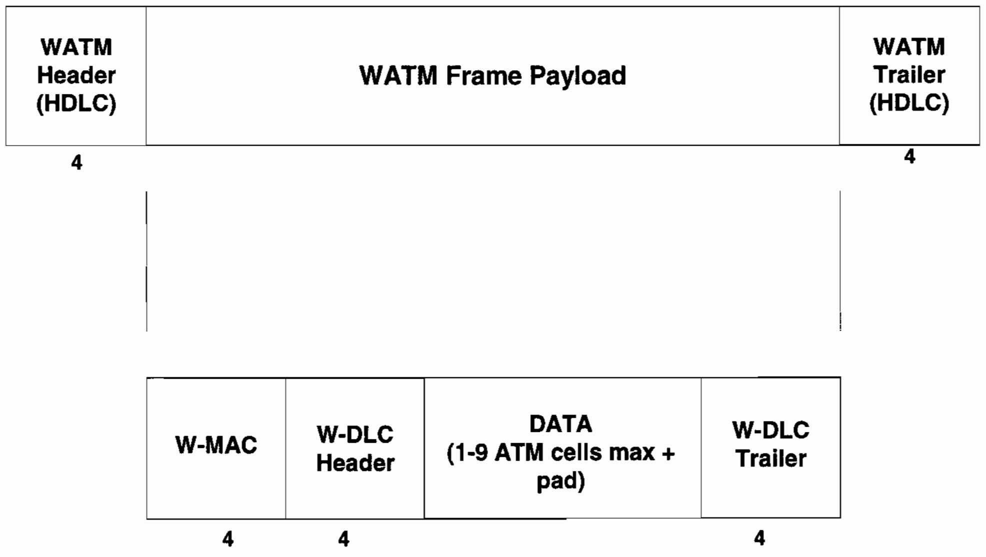


Figure 4: WATM Frame Format

1.2.3.2.3 Wireless Multi-path Routing Protocol (WMRP)[5] module:

The protocol as proposed by Fadi Wahhab is a multiple path extended distance vector algorithm that utilizes information regarding the length and second-to-last (predecessor) of the shortest path to each destination to prevent loops. To reduce routing overhead, updates are sent only after a topology change, and to ensure connectivity, Hello Packets are exchanged between neighboring nodes every Hello Interval. If no such packets are received at a certain node from its neighbor for three consecutive Hello Intervals, the node assumes that connectivity with that neighbor has been lost. The protocol provides multiple paths to each destination, makes use of the channel nature of RDRN network to send different information to different neighbors (split horizon), which increases the route availability, and sends changes in routes to the neighbors affected by that change.

The routing table of node x contains an entry for each destination j, each of those entries contain the following:

- The destination identifier (j).
- A list of neighbors, if any, that would be affected by the change and need to be informed about it.
- One or more path(s) information, each path is identified by a different neighbor and contains the following information:
- The identifier of the next hop (k).
- Metric representing the distance to j.
- The identifier of the predecessor (next to last).

A detailed description of the protocol is given in Chapter 3.

The following example gives a brief illustration of how the availability of multiple paths to each destination helps in maintaining the connectivity in a highly dynamic environment like the RDRN.

Example 1

For the sake of simplicity the example shows how the protocol works for only one destination (Node J). The table by a node in Figure 5 represents the routing table at that node for the destination, Node J. Each row in the table gives the destination - next hop – hop count – second-to-last (in the same order).

Consider the network shown in Figure 5. Here we would not analyze as to how the routing table at each node was created; and we will assume that the network has stabilized.

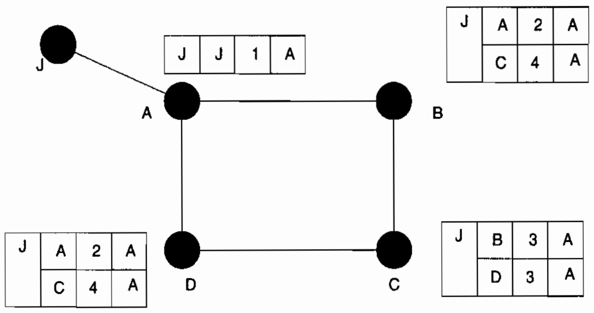


Figure 5: Routing Scenario, Example 1[5]

Say now that node B is sending data to node J (which takes the path B-A-J) and node B moves as shown in the Figure 6. As soon as node B detects the link failure between A and B, it starts using another route to destination J which it already has in its routing table i.e. routing the data through node C. The link failure will be detected after three Hello intervals, unless a lower layer protocol can detect such a failure and provide feedback to the routing agent earlier. Node B also updates its routing table, and informs node C about it. Node C will also update its routing table and inform node D about it with the next Hello packet. The idea from this example is to show that the data would still be routed correctly before the network converges. Figure 6 shows the routing entries at each node for destination J after the network converges.

Note that the amount of data packets dropped is minimal, because the routers keep multiple paths to a destination so the nodes can make use an alternative routes even before the network converges.

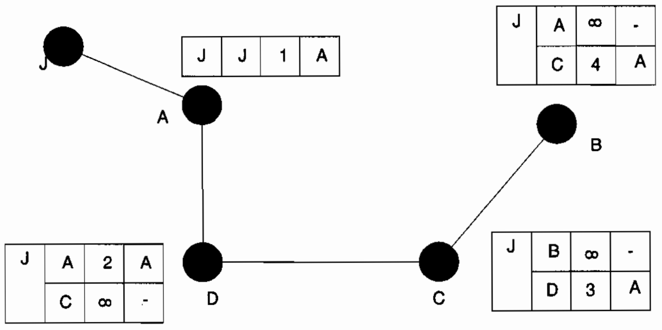


Figure 6: Changed Routing Scenario, Example1[5]

1.3 Requirements

Since, one primary goal of this work was to do a comparative evaluation of IP vs ATM and to look into the scalability and the performance of the network controller; the first step in the design of the emulation environment was to isolate the actual radios (radio controller) and to provide an alternate mode of connectivity for both the orderwire[1] and the high-speed link. This implied that there should be a mechanism for emulating the beams over the ATM testbed (shown in Figure 7) and the ability to multiplex/de-multiplex traffic for different destinations over the same beam.

Another requirement was to implement the routing protocol (WMRP) as it had not been done yet, and to integrate it with the other software modules of the RDRN.

Hence, the objectives of this work were:

- Design and implement a configurable protocol stack that emulates the high-speed link. By configurable, it means that the protocol stack might be from a set of valid combinations like
  - SAR+DLC (Segmentation & Re-assembly + Data Link Control)
  - DLC (Data Link Control)
  - SAR + QoS (Quality of Service) + DLC

  This would help to identify what possible combinations give better performance. The fact that the protocol stack would emulate the beams on the ATM testbed did not preclude that it should not work on the actual radios.
- Implement the WMRP and integrate it with other software modules.
- Make the required minimal changes in the existing software modules to adapt it to the emulation environment.
- Run different scenarios in the emulation environment and measure different performance metrics.

[1] Implemented by Leon Searl

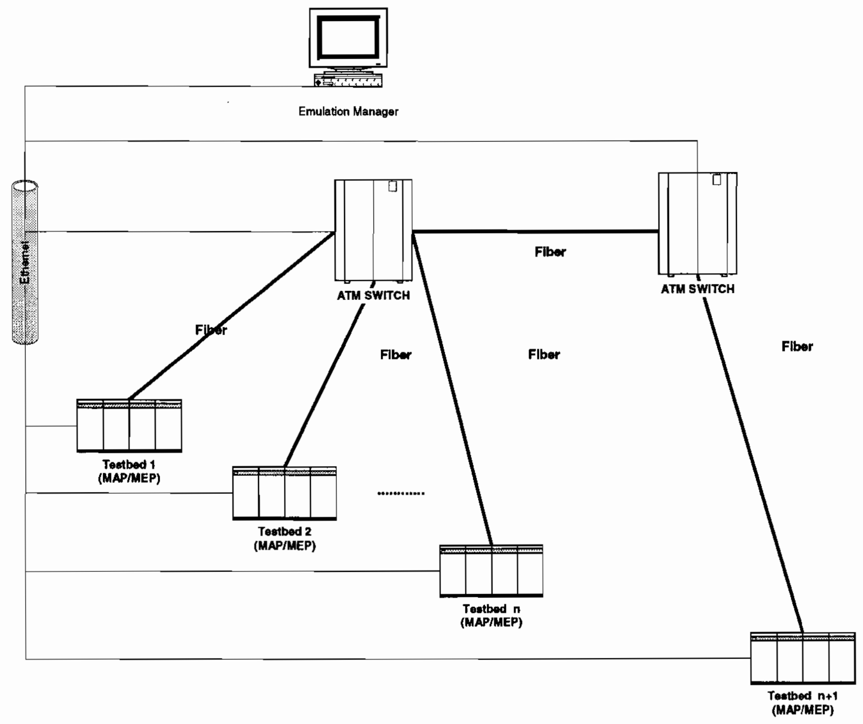


Figure 7 Physical Connectivity of the Testbeds

A reasonable man adapts himself to suit his environment. An unreasonable man persists in attempting to adapt his environment to suit himself. Therefore, all progress depends on the unreasonable man.

George Bernard Shaw

Chapter 2

Requirements and Design of the Emulation Environment

In this chapter, the requirements of the emulation environment are identified by taking a given scenario into consideration and analyzing how it would behave in the "field". Once the requirements are identified, possible solutions for those requirements are discussed and then the same scenario is considered in the proposed emulation environment.

2.1 Configuration of the Testbeds

Before we look into the requirements and the solutions for those requirements, we would look as to how the testbeds are physically connected. The physical connectivity of the ATM-testbeds is as shown in Figure 8.

The characteristics of the layout shown in Figure 8 are:

- There are n+1 (equals 24) testbeds, which are Linux boxes with Red-Hat 4.2 running on them. The Linux kernel is 2.2.1 with ATM tools ver 0.53. Each of these represents a MAP or a MEP.
- There is a master machine called the Emulation Manager.

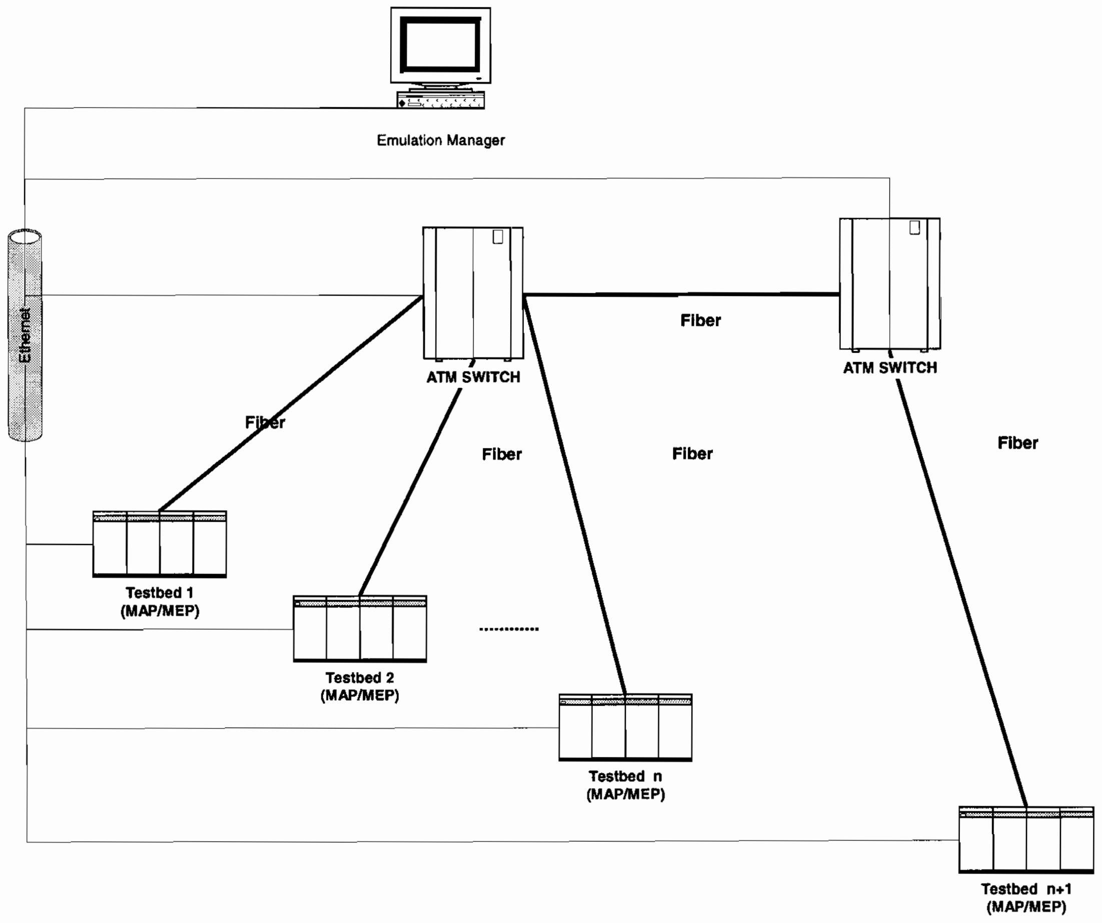


Figure 8: Physical Connectivity of the Testbeds

- Each testbed has 100Mbps Ethernet connectivity so that it is possible to establish remote-login.
- Each of the testbed has an ENI-155p ATM-card, which is connected through a fiber to the FORE-ATM (asx200bx) switch with S_ForeThought_5.3.1 FCS-Patch. The FORE-ATM switches have 12 ports. This implies that at any given time only 12 machines can be connected to the switch
- If the emulation scenario contains more than 12 nodes, then an another FORE-ATM switch could be used.

2.2 Identification of Requirements and Solutions to those Requirements

Let us consider the following scenario as shown in Figure 9 in which we have 2 MAPs & 2 MEPs and Node D is moving westwards. For the sake of simplicity, we assume that all the nodes come up at the same time.

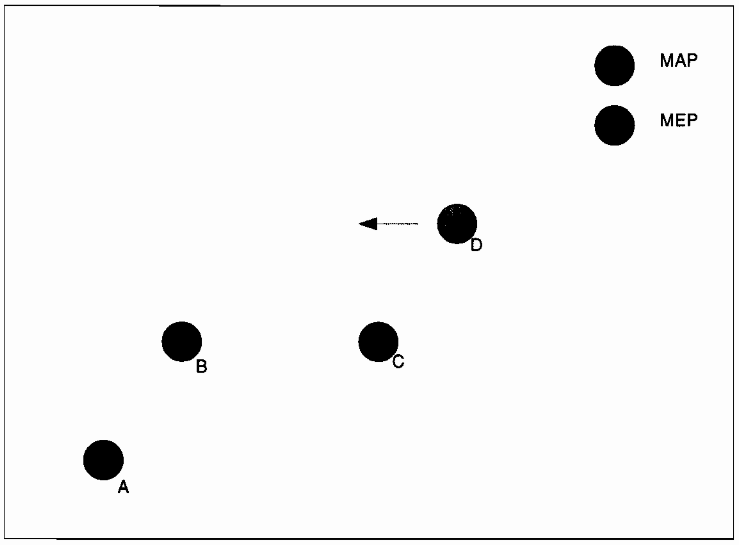


Figure: 9 Four-Node Scenario

We will now go through the steps that are involved before any data transfer can take place between any two nodes that are *not* neighbors. This would help us to identify the various components that need to be emulated.

The different stages involved in the field are as follows:

2.2.1 Stage 1: Exchange of Information over the Orderwire

As soon as the nodes come up they obtain their position from the GPS receiver and start transmitting their position over the orderwire. All those nodes that are within the orderwire range would hear the location. As shown in the Figure 10, Node A receives the orderwire information from Node B, Node B receives the orderwire information from Node A & Node C, Node C receives the orderwire information from Node B & Node D and Node D receives the orderwire information from Node C.

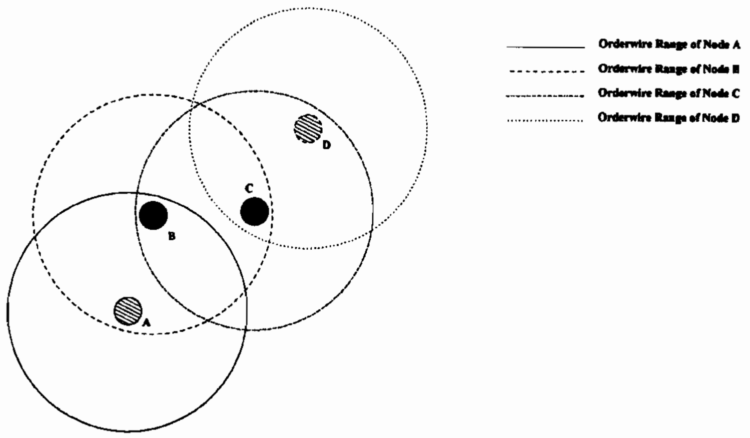

Orderwire Range of Node A
Orderwire Range of Node B
Orderwire Range of Node C
Orderwire Range of Node D
A
B
C
D

Figure 10: Orderwire Range for the 4 Node Scenario

Requirement:

Req.1: In the emulation environment there would not be any GPS receiver from which the nodes would be able to get their positioin at different instances of time. Hence, the emulation environment should provide a mechanism by which the nodes would know of their GPS values (locations) at different instances of time. This in turn would help to emulate the mobility of the nodes.

Solution[7]:

The orderwire module was be modified so that now instead of interfacing with an actual GPS receiver, it opens a socket to the Emulation Manager on a predefined port and listens on that port for its location information from the Emulation Manager. The socket type is of User Datagram Protocol (UDP) type. The value of the predefined port is 20361. The Emulation Manager sends NMEA 0183 format GPS message to each of the nodes and the entire GPS message is sent in a single datagram. The GPS data is sent to each of the node every 1.8 seconds. Detailed information about the GPS data format can be found at http://www.ittc.ukans.edu/~searl.

Req.2: It should also provide a way by which the orderwire packets are broadcasted to all the other nodes which are within the orderwire-range of the given node.

Solution:

The orderwire module was modified so that now instead of interfacing to the packet radios to transmit the orderwire packet, it opens a socket to the Emulation Manager over the Ethernet on a pre-defined port and sends & receives the orderwire packets on that port. The socket type is of User Datagram Protocol (UDP) type. The value of the predefined port is 20362. The Emulation Manager reads Orderwire datagrams on its well-known port and then re-transmits the same datagram to zero or more of the nodes depending on the topography and node separation. The nodes receive the Orderwire UDP datagrams on the same well-known port number as that of the Emulation Manager. The Emulation Manager does not look into the contents of the packet so it would not know the current position advertised by the nodes and hence it is not able to calculate the distances between them. This implies that the Emulation Manager would not be able to filter the orderwire packets and each node in the network would get the orderwire packets from all the other nodes. This is obviously different from what happens in the field, but it is of no concern since the orderwire would never attempt to configure a high-speed link to the nodes that are *not* within its range.

2.2.2 Stage 2: Establishing the Network Topolgy & High-Speed Connectivity

After getting the position of the other nodes which are within its orderwire range, each node would execute the topology algorithm and depending on whether it is a MAP or a MEP, it would try to establish the high-speed link with the neighbors. It is quite possible that a node might not hear from all the nodes within its orderwire-range because of the topography (terrain blocking).

The topology algorithm works differently on the MAPs & the MEPs. On the MAPs, the algorithm calculates the distance between itself and all the other nodes it has heard from. Since the range of the orderwire is different from that of the high-speed link (for simplicity, we will assume that they are the same), the topology algorithm checks whether the calculated distances are within the range for high-speed connectivity or not. If it so, it tries to establish point-point links with the nearest four, each being on a different beam. On the MEPs, the algorithm calculates the distance between itself and all the MAPs it has heard from. If the above-calculated distance is within the range for high-speed connectivity, it tries to establish the point-to-point link with the *nearest* MAP. The scenario after the execution of the topology algorithm and the beam used by the nodes has been shown in Figure 11.

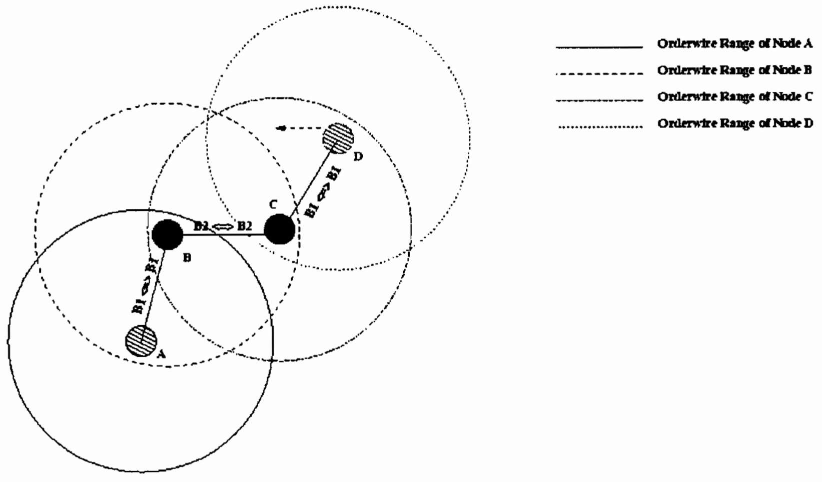

Orderwire Range of Node A
Orderwire Range of Node B
Orderwire Range of Node C
Orderwire Range of Node D
A
B
C
D
B1<=>B1
B1<=>B2
B1<=>B1

Figure 11: High Speed Connectivity for the 4 Node Scenario[2]

[2] The text next to the links indicate the beam # being used by the nodes.

Though the point-to-point connections has been established between the neighbors, data transfer can take place only between neighbors but *not* between any two nodes which are *not* neighbors of each other. This would be possible after Stage 3.

Requirement:

Req.3: Since the emulation environment would be implemented on ATM-testbeds the environment should provide a way to emulate the beams.

Req.4: Moreover, since the traffic for more than one destination might be going on the same beam there should be a mechanism for multiplexing the traffic at the sources and de-multiplexing the traffic at the destinations and the intermediate nodes.

Solution:

As outlined in Chapter 1, a MAP can establish a maximum of 4 point-to-point high-speed links and a MEP can establish only one. One solution to this problem would be to have 4 ATM cards, where each card would represent a single beam. This solution is neither elegant nor feasible since it would imply that each machine would take up 4 ports on the FORE-ATM switch. Hence, the solution to the above problem is to design and implement something called "Virtual ATM (VATM)" shown in Figure 12. This creates 4 ATM devices on top of a single ATM card but the RDRN software would see as if the given machine has 4 ATM Network Interface Cards. Each of these VATMs would have a different interface id (ITF). These VATMs would be hooked to the underlying ATM card on a given VCI, called the physical VCI. However, this physical VCI would be transparent to the higher protocol layers (like Classical IP). They would be sending the traffic to different destinations on what is called the logical VCI. This way it would also be possible to multiplex traffic for different destinations on the same beam and a corresponding de-multiplex at the destinations or the intermediate nodes.

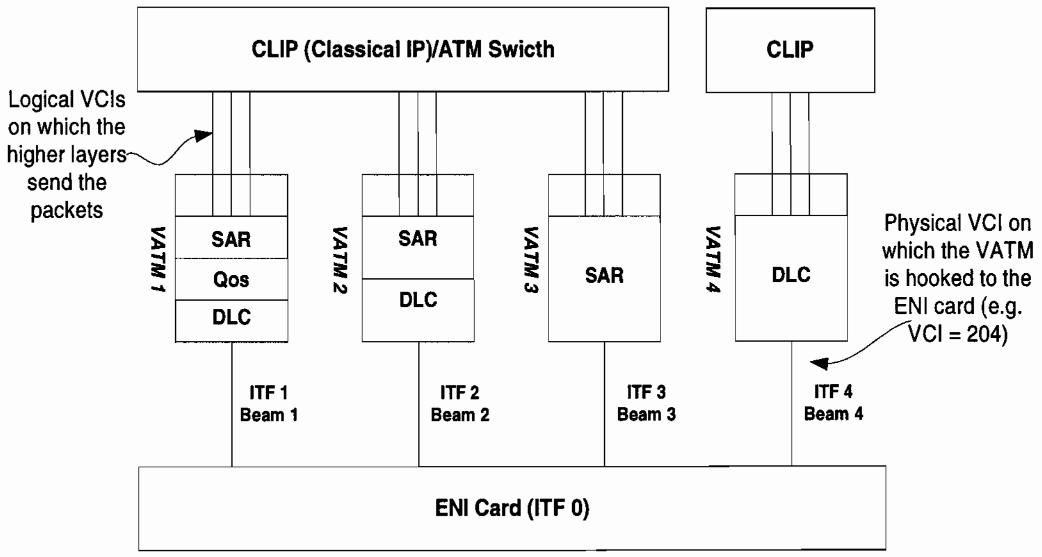

CLIP (Classical IP)/ATM Swicth
CLIP
Logical VCIs on which the higher layers send the packets
VATM 1
SAR
Qos
DLC
VATM 2
SAR
DLC
VATM 3
SAR
VATM 4
DLC
Physical VCI on which the VATM is hooked to the ENI card (e.g. VCI = 204)
ITF 1 Beam 1
ITF 2 Beam 2
ITF 3 Beam 3
ITF 4 Beam 4
ENI Card (ITF 0)

Figure 12: VATM Architecture

Req.5: The Emulation Manager should also be able to establish/tear-down high-speed connectivity between the neighbors.

Solution:

As it was shown in Figure 8 each of the nodes (testbed) are connected to the FORE-ATM switch. Hence, to establish ATM connectivity between two testbeds the PVCs need to set up on both the testbeds as well as the FORE-ATM switch. The PVCs on the testbeds would be set up by the Orderwire using the Linux-ATM tools. To set up the PVCs on the FORE-ATM switch, the node (Orderwire) would open a UDP Internet socket connection to the Emulation Manager on a well-known port (20363) over the Ethernet and send a request to create the PVC on the switch. The Emulation Manager[7] on receipt of such a request would send an SNMP (Simple Network Management Protocol) request to the FORE-ATM switch to create the PVC. For the connection between two nodes to be complete, both the nodes need to make such a request. Similarly, when the high-speed connection between to nodes need to be torn-down because they went out of range or because of the topography, the nodes need to send a request to the Emulation Manager to delete the PVCs on the FORE-ATM switch.

2.2.3 Stage 3: Creation/Exchange of Routing Information

Once the point-to-point connectivity is established between the neighbors, the nodes add their neighbors in their routing table and inform the other neighbors about the nodes which they have added in their routing table. This way the information about all the nodes propagates through the network, so every node comes to know of all the other nodes and the routes to those nodes. A detailed description of how the routing table is created and what information is exchanged between neighbors is given in Chapter 3.

Requirement:

Req.6: The routing protocol needs to be implemented as it has *not* been done so yet.

Solution:

To reduce the complexity of implementation, the routing protocol would run in the user-space but it would use Netlink sockets to change the kernel routing table as and when required. The routing protocol would also run on a pre-defined port (55555). To exchange the Hello Packets and the routing table, the nodes would open a TCP socket connection on the above port over the high-speed link. In this particular case, TCP is used instead of UDP since the WMRP expects guaranteed delivery of all the packets it sends and if it is not able to do so it would take that as indication of the link going down. This might lead to the high-speed connection getting torn-down.

2.3 Components of the Emulation Environment

Now we will consider the same network as we did in Section 2.1 in the emulation environment. The emulation environment would now consist of the Emulation Manager, 4 testbeds that would be connected to the FORE-ATM switch through the fiber and would also have Ethernet connectivity. The following section explains the basic functionality expected of the various components (as shown in Figure 13) before we analyze the network under the emulation environment.

2.3.1. Emulation Manager[3]

The various components of the Emulation Manager would be as follows:

- Runtime Manager:

  The functions performed by the Runtime Manager are:

  i. Send the GPS data to each node for node motion and location
  ii. Broadcast the Orderwire Packets to all the nodes except from which it received the packet.
  iii. Creation/Deletion of PVCs on the FORE-ATM switch. The PVCs on the switch would be created when a request comes from the node. The PVCs on the switch would be deleted when such requests come from the nodes or when the runtime manager itself realizes that the nodes are no longer with the high-speed connectivity range.

- Scenario File

  The scenario file contains information regarding the nodes that are there in the network and also their positions at different instances of the emulation time. *The format of the scenario file can be found in Appendix C.2*

- Port-Map File

  The port-map file contains information as to which port the testbeds are connected to on the FORE-ATM switch. This information is required to create/delete the PVCs on the FORE-ATM switch. *The format of the port-map file can be found in Appendix C.1*

[3] All the functionalities expected from the Emulation Manager was implemented by Leon Searl. Detail information about the design is available at http://www.ittc.ukans.edu/~searl.

2.3.2 Modules on the Node

The different modules running on the nodes would be as follows:

- Orderwire Module
- Routing Module
- VATM driver

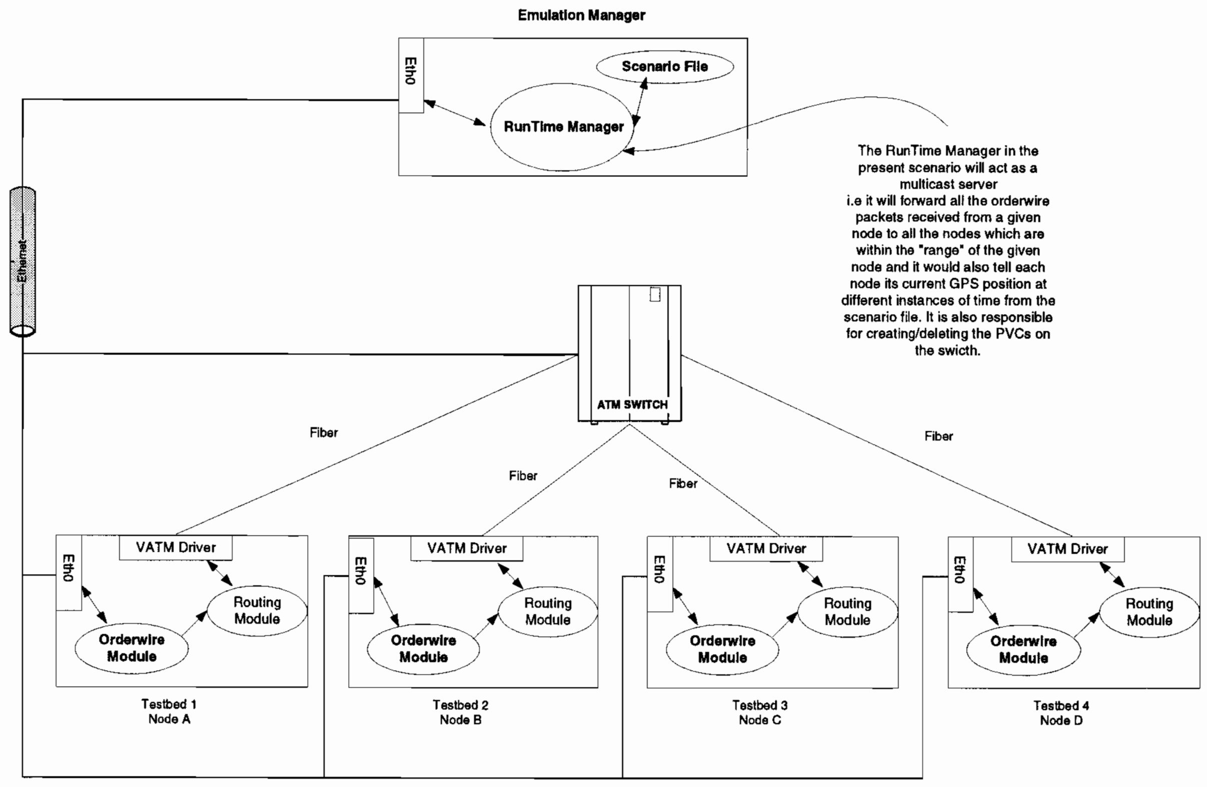

Emulation Manager
Eth0
Scenario File
RunTime Manager
The RunTime Manager in the present scenario will act as a multicast server i.e it will forward all the orderwire packets received from a given node to all the nodes which are within the "range" of the given node and it would also tell each node its current GPS position at different instances of time from the scenario file. It is also responsible for creating/deleting the PVCs on the swicth.
Ethernet
ATM SWITCH
Fiber
Fiber
Fiber
Fiber
VATM Driver
Routing Module
Orderwire Module
Testbed 1 Node A
Testbed 2 Node B
Testbed 3 Node C
Testbed 4 Node D

Figure 13: Software Modules in the Emulation Environment

2.3.2.1 VATM Driver

Before the emulation is started, the VATM module would be used to create 4 ATM devices on the MAP and 1 on the MEP. Each one them would be configured with the desired protocol stack and the CBR for the physical VCI. Each one these would represent a beam and have a different ITF as shown in Table 1.

VATM #	ITF #	Physical VCI
vatm1	1	201
vatm2	2	202
vatm3	3	203
vatm4	4	204

Table 1: Physical VCI/ITF for the VATM

The VATM has been described in detail in Chapter 3.

2.3.2.3 Orderwire Module

The orderwire module includes the emulated GPS receiver that would receive information about its location from the Emulation Manager on the Ethernet. The orderwire would use these GPS data in its orderwire packet and send it over to the Emulation over the Ethernet.

When the orderwire module receives orderwire packets from other nodes, it calls the topology algorithm and if a high-speed point-to-point connectivity needs to be established it creates the logical PVC to the desired destination. The logical PVC would be created on the beam (vatm) allotted by the topology algorithm. The orderwire module also sends a request to the Emulation Manager to create the PVC on the switch between the ports on which the two machines are connected. This PVC on the switch is used to connect the physical VCIs. The Orderwire Module also informs the routing module about the node (and the beam #) to which it has established the high-speed link.

The orderwire module would delete the logical VCIs if it does not hear from the nodes to which it had established high-speed connectivity within a specified timeout. It would also send a request to the Emulation Manager to delete the physical PVC on the switch between the two nodes.

2.3.2.4 Routing Module

Once the high-speed link has been established between the two nodes, the routing protocol would start exchanging the Hello Packets over the high-speed link. The Hello Packets would be sent on a TCP socket connection as the routing protocol expects guaranteed delivery. Once the 3 Hello Packets have been exchanged within a specified interval, both the neighbors would exchange their routing table. This way information about all the nodes and their routes would propagate across the network. After the routing table has been exchanged, the two nodes would send Hello Packets every Hello Interval, which is of the order of seconds. If routing module does not hear from its neighbor within a specified time, it would delete the high-speed connection to that node i.e. it would delete the PVC to the node but it would not send the request to the Emulation Manager to delete the PVC on the switch. This would be taken care by the Orderwire. The reason for the above is that the routing module would be able to detect the link failure much faster than the Orderwire. The routing protocol and its implementation is described in detail in Chapter 3.

2.3.2.5 An Example Network

Now we will consider the scenario as outlined in Section 2.1 at different emulation time under the above designed emulation environment. Let's say that the all the nodes in the network come up at t=0 i.e. the state of the network is as shown in Figure 14.

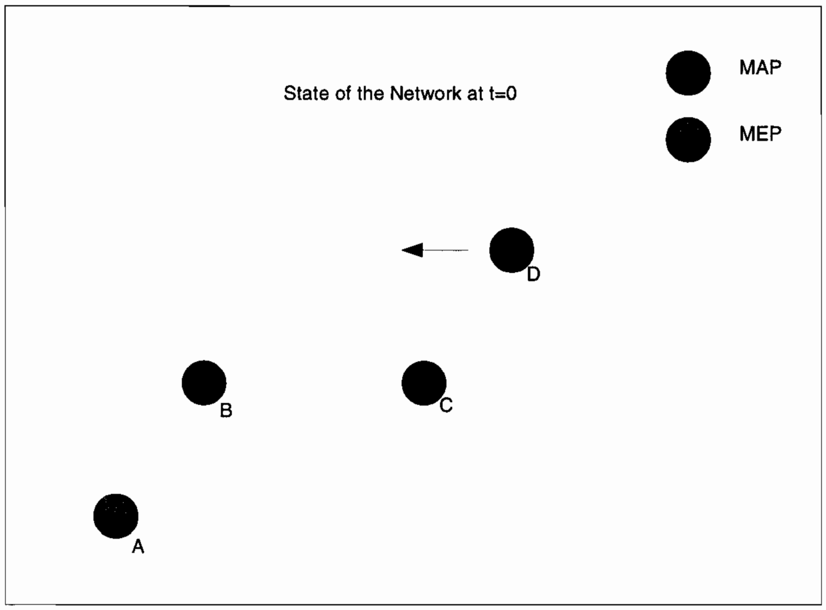


Figure 14: 4 Node Scenario

Node D is moving westwards and we will assume that at t=t+Δt_5 it would go out of range of Node C and come within the range of Node B. The state of the network at t=t+Δt_3 & at t=t+Δt_5 is shown in Figure 15 & Figure 16.

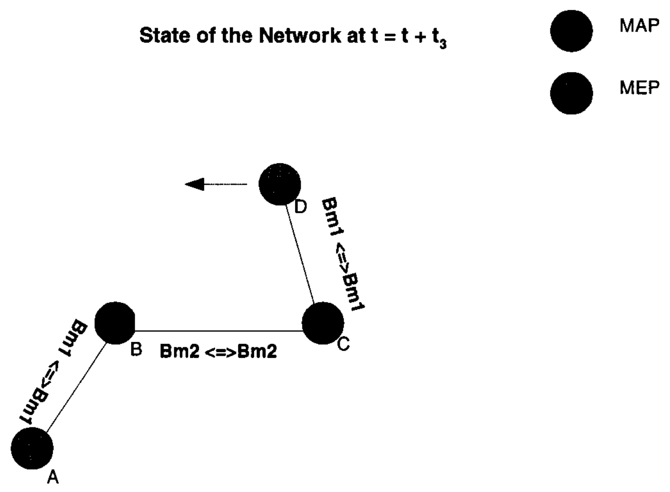


Figure 15: State of the Network at t = t + Δt_3

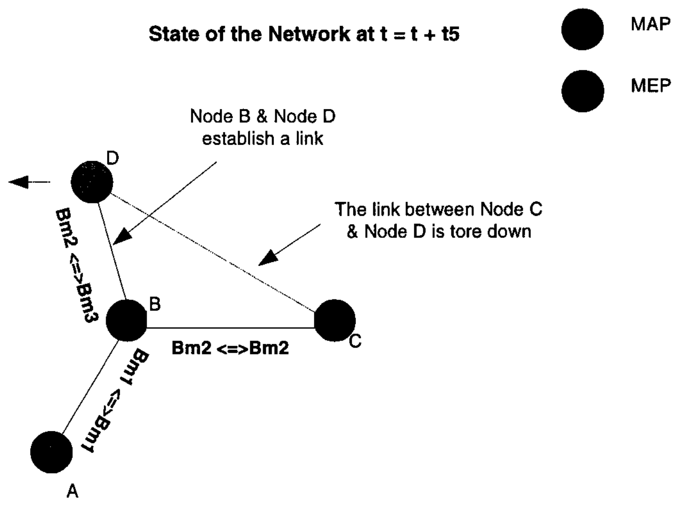

State of the Network at t = t + t5
MAP
MEP
Node B & Node D establish a link
D
The link between Node C & Node D is tore down
Bm2 <=>Bm3
B
Bm2 <=>Bm2
C
Bm1 <=>Bm1
A

Figure 16: State of the Network at t = t + Δt_5

The sequence of events/commands that would happen at the nodes has been outlined in Table 2.

Emulation Time	Node A	Node B	Node C	Node D
t = 0	Nodes start receiving their GPS location from the Emulation Manager			
$t = t + \Delta t_1$	Nodes start transmitting the orderwire packets to Emulation Manager that forwards all orderwire packets to all the nodes, except to the one it received the packet from.			
$t = t + \Delta t_2$	The topology algorithm is invoked on all the nodes. The algorithm would decide as to which node connects to which and on what beam. Here we assume that all the nodes have received at least one orderwire packet from all the other nodes.			
	Node A decides to connect to Node B on its Beam 1 (vatm1)	Node B decides to connect to Node A on its Beam 1 (vatm1) & to connect to Node C on its Beam 2 (vatm2)	Node C decides to connect to Node B on its Beam 2 (vatm2) & to connect to Node D on its Beam 1 (vatm1)	Node D decides to connect to Node C on its Beam 1 (vatm1)
$t = t + \Delta t_3$	• Node A creates the PVC to Node B on vatm1 • Requests the EM to establish a PVC between Node A & Node B using the physical VCI associated with vatm1 • Informs the routing module about the new neighbor Node B	• Node B creates the PVC to Node A on vatm1 & to Node C on vatm2 • Requests the EM to establish a PVC between Node B & Node A using the physical VCI associated with vatm1 and between Node B & Node C using the physical VCI associated with vatm2 • Informs the routing module about the new neighbors Node A & Node C	• Node C creates the PVC to Node D on vatm1 & to Node B on vatm2 • Requests the EM to establish a PVC between Node C & Node D using the physical VCI associated with vatm1 and between Node C & Node B using the physical VCI associated with vatm2 • Informs the routing module about the new neighbors Node B & Node D	• Node D creates the PVC to Node C on vatm1 • Requests the EM to establish a PVC between Node D & Node C using the physical VCI associated with vatm1 • Informs the routing module about the new neighbor Node D
$t = t + \Delta t_4$	All the neighbors are able to exchange the 3 Hello Packets within the specified time and exchange their routing table. This way all the nodes in the network know about the existence of the other nodes and the different possible routes to them.			
$t = t + \Delta t_5$	Not Applicable	Node B realizes through the orderwire that Node D is now within its range and hence establishes high-speed connectivity to Node D. The orderwire then informs the routing module about the new neighbor & the beam # associated with it.	The EM deletes the PVC on the switch between Node C & Node D as it realizes that they are out of the high-speed range. The routing protocol detects this link failure and deletes the PVC to Node D on its end. If it would have an another route to Node D, it would install that.	The EM deletes the PVC on the switch between Node C & Node D as it realizes that they are out of the high-speed range. The routing protocol detects this link failure and deletes the PVC to Node C on its end. If it would have an another route to Node C, it would install that.

$t = t + \Delta t_6$	Detects the new route to Node D	Node B successfully exchanges the 3 Hello Packets and hence exchanges its routing table with Node D. It also propagates the new route to Node D through its neighbors.	Detects the new route to Node D	Node D successfully exchanges the 3 Hello Packets and hence exchanges its routing table with Node B.

Table 2: Sequence of Events in the Emulation Environment

Never tell people how to do things. Tell them what to do and they will surprise you with their ingenuity.

George S. Patton, Jr.

Chapter 3

Implementation

In this chapter, the Virtual ATM and the Wireless Multi-Path Routing Protocol has been described in detail. The implementation of the above are also discussed

---

3.1 VATM (Virtual ATM):

The VATM on ATM/Ethernet is a driver that provides multiple logical ATM interfaces on a single ATM/Ethernet card. The purpose of the above mentioned driver is to provide an environment in which the RDRN software believes that there are multiple ATM Network Interface Cards on the machine.

Each of these logical interfaces represent a beam and these interfaces are hooked to the ATM card on a given “physical” VCI of AAL5 type. The VATM architecture is shown in Figure 17. Moreover, the VATM provides a mechanism for multiplexing the traffic for various destinations over the same physical VCI. The traffic for different destinations is sent on different “logical” VCIs. This provides the ability to emulate the TDMA that is required when a given Mobile Access Point (MAP) is communicating with a set of Mobile End Points (MEP) over the same beam. It also enables the MEP to send traffic for various destinations over the same beam.

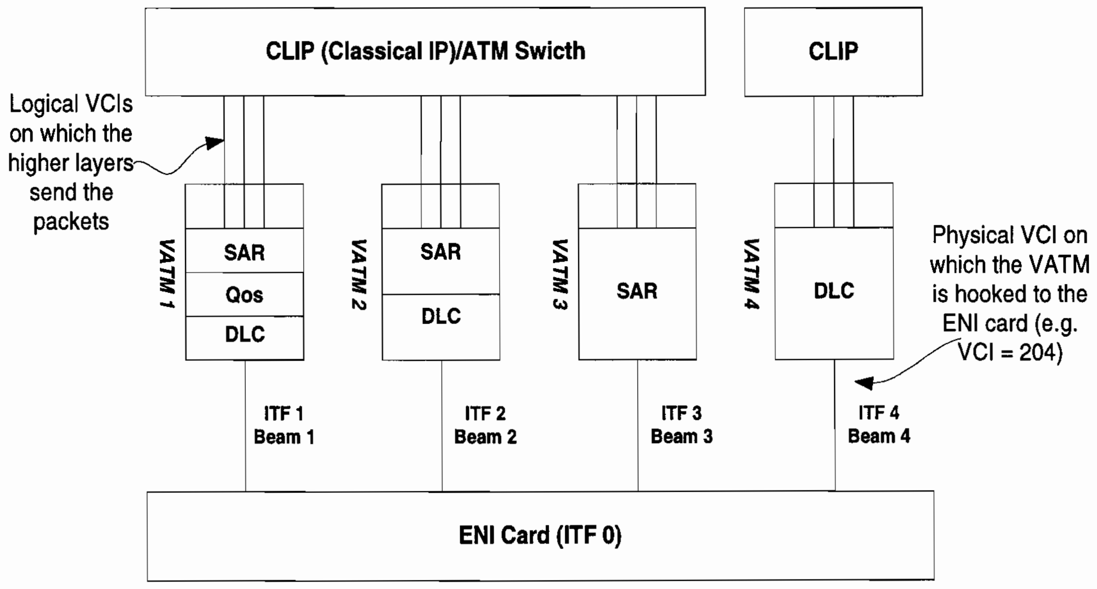

CLIP (Classical IP)/ATM Swicth
CLIP
Logical VCIs on which the higher layers send the packets
VATM 1
SAR
Qos
DLC
VATM 2
SAR
DLC
VATM 3
SAR
VATM 4
DLC
Physical VCI on which the VATM is hooked to the ENI card (e.g. VCI = 204)
ITF 1 Beam 1
ITF 2 Beam 2
ITF 3 Beam 3
ITF 4 Beam 4
ENI Card (ITF 0)

Figure 17: VATM Architecture

3.1.1 Protocol Stacks on the VATM

The VATMs that are created on a given ATM card can have different protocol stacks. The different possible combinations of the protocol stacks are outlined in the following section. However, in each of the following sections the AAL (ATM Adaptation Layer) has not been shown, as it is common to all of them as it provides a hook to the Linux-ATM architecture.

3.1.1.1 SAR (Segmentation & Re-assembly):

In this case the packets coming from the higher layer (CLIP) are segmented to 53 bytes (5 bytes header + 48 bytes of data) of ATM cells and sent down to the ATM driver which packets the cells in an AAL5 frame. This is shown in Figure 18.

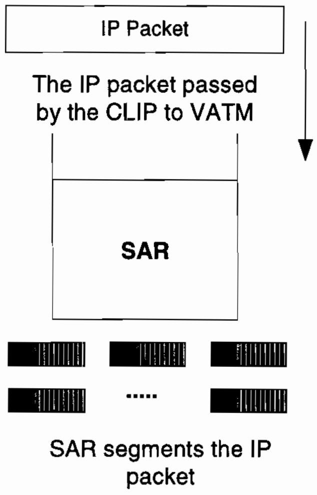

IP Packet
The IP packet passed by the CLIP to VATM
SAR
.....
SAR segments the IP packet

Figure 18: VATM with SAR layer

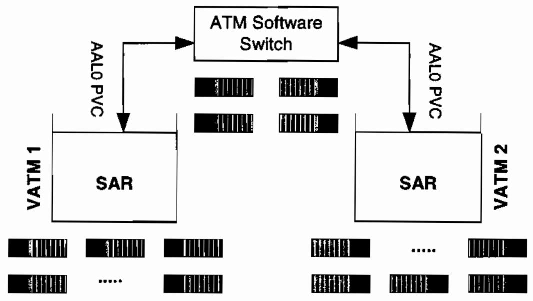


Figure 19: VATM (with SAR) hooked to the Micro-Switch

The above VATM can be hooked to the ATM software switch. In this case the switch would open an AAL0 PVC to the VATM and hence the SAR would not perform the re-assembly of the cells. It would pass the ATM cells to the switch that does switching depending on the entries in its switching table. This is shown in Figure 19.

3.1.1.2 SAR + DLC:

In this case the packets coming from the higher layer (CLIP) are first segmented to 53 bytes (5 bytes header + 48 bytes of data) of ATM cells and then sent to the DLC layer. The DLC layer puts a given number of ATM cells in the DLC packet, which has a default value of 7 when the VATM is created. However, the number of ATM cells in a DLC packet can be changed on the fly. The DLC layer sends the DLC packet down to the ATM driver that sends the DLC packets as AAL5 packets. This is shown in Figure 20.

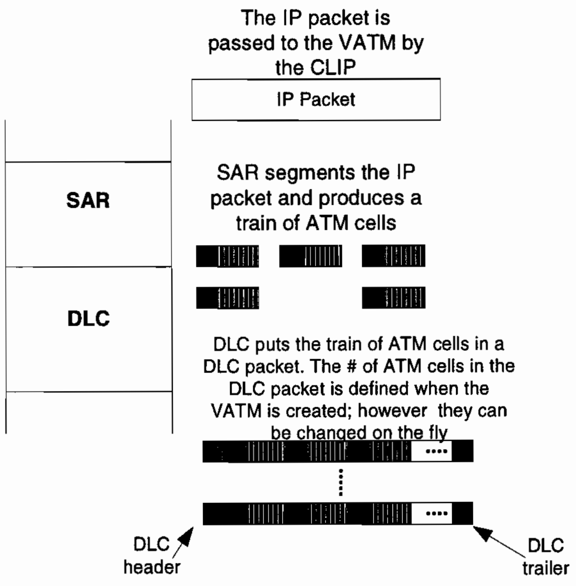

The IP packet is passed to the VATM by the CLIP
IP Packet
SAR
SAR segments the IP packet and produces a train of ATM cells
DLC
DLC puts the train of ATM cells in a DLC packet. The # of ATM cells in the DLC packet is defined when the VATM is created; however they can be changed on the fly
DLC header
DLC trailer

Figure 20: VATM with SAR+DLC Layer

In this case also, the VATM can be hooked to the ATM switch as shown in Figure 19.

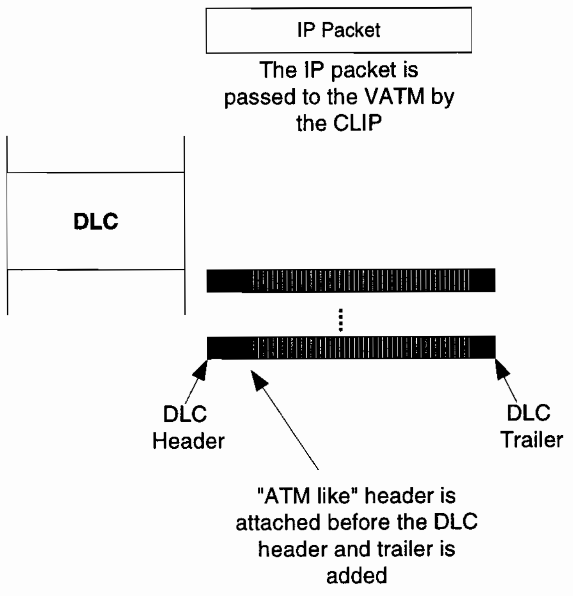
3.1.1.3 SAR + QoS +DLC

In this case the packets coming from the higher layer (CLIP) are passed to the SAR which does the segmentation of the packets into 53 bytes of ATM cells and passes the train of cells to the QoS[4] layer. The QoS layer maintains different queues for traffics of different priority and depending on its scheduling algorithm, sends the cells to the DLC layer which then creates a DLC packet with a given number of the ATM cells. The DLC packet is passed to the ATM driver which transmits the DLC packets as AAL5 packets.

[4] Implemented by Saravanan Radhakrishnan

3.1.1.4 DLC

In this case the packets coming from the higher layer (CLIP) are passed to the AAL_DLC_GLUE_LAYER which attaches a 5 byte ATM like header and passes the packet to the DLC layer. The 5 byte ATM-like header is added to store the logical VCI values. As explained above, this would help in multiplexing and de-multiplexing the traffic for different destinations over the same VATM (beam). The DLC layer attaches its own header and trailer and sends the packet down to the ATM driver that sends the DLC packets as AAL5 packets. If the size of the IP packet passed from the CLIP along with the 5 byte ATM like header and the DLC header & DLC trailer would be greater than that supported by the ENI driver, then the AAL_DLC_GLUE_LAYER would do segmentation of the packet passed from above. This is required because the IP over ATM (CLIP) specification says that the MTU should be no larger than 9180 bytes, hence there are times when the CLIP would pass a packet of the above size to the DLC layer. Hence, when the DLC would add its own header and trailer, it would cause an overflow on the ENI card. A corresponding re-assembly would be done at the other end. However, in this case the ATM software switch cannot be hooked to the VATM. This has been shown in Figure 21.



IP Packet
The IP packet is passed to the VATM by the CLIP
DLC
DLC Header
DLC Trailer
"ATM like" header is attached before the DLC header and trailer is added

Figure 21: VATM with DLC Layer

3.1.2 Data Structures for VATM

The information about each of the VATM is stored in a structure called the vdevice. The structure is as shown below:

```
struct vdevice
      {
      int vatm_itf;                     /* virtual ATM interface */
      enum vdev_type type;              /* vdev type */
      void * dev;                       /* the real device (e.g.,eth0 or ATM itf 0) */
      struct rca_info * rca_info;
      struct packet_type * pt;          /* packet encapsulation type */
      struct vdev_stats stats;
      struct vdevice * next;
      unsigned char * my_mac;           /* for ethernet, its the local mac address */
      unsigned char * peer_mac;         /* for ethernet, its the peer */
      unsigned int vci;                 /* for atm, its the associated vci */
      struct atm_vcc * vcc;
      void * top_layer;                 /* top layer */
      void * bottom_layer;              /* bottom layer */
};
```

The `vatm_itf` contains the Itf assigned to the VATM. The `vdev_type` field specifies whether the VATM is built on ATM or Ethernet. The valid values are:

```
enum vdev_type {
        VDEV_RAW = 0,
        VDEV_MAC = 1,
        VDEV_MAC_RCA = 2,
        VDEV_ATM = 3
};
```

The `VDEV_RAW`, `VDEV_MAC` and `VDEV_MAC_RCA` are the operating modes on the Ethernet; where `VDEV_RAW and VDEV_MAC_RCA` are the operating modes specific to the RDRN radio. `VDEV_ATM` is used when the VATM is created on the ATM.

The `(void) *dev` points to the actual device on which the VATM is built. If the VATM is built on the ATM, it would point to `struct atm_dev` and if built on Ethernet it would point to the `struct device`. The `struct rca_info` is used to store information about the credits that is used for the RDRN radios. The `vdev_stats` is a structure that stores statistical information about each of the VATM. The fields of it are as follows:

```
struct vdev_stats
{
        int tx_sent;
        int tx_dropped;
        int rx_received;
        int rx_dropped;
};
```

The my_mac and peer_mac fields are used when the virtual device is created on the Ethernet. The vci is used to store the physical VCI on which the VATM is hooked to the ATM card. The (void) * top_layer and the (void) * bottom_layer point to the top and the bottom layer on the protocol stack on the VATM.

The structure atm_vcc is used to store all the device-independent parameters for the physical VCI on which the VATM is hooked to the ATM card. Some of the elements of the struct atm_vcc are:

```
struct atm_vcc {
        unsigned short  flags;            /* VCC flags (ATM_VF_*) */
        unsigned char   family;           /* address family; 0 if unused */
        short           vpi;              /* VPI and VCI (types must be equal */
        int             vci;
        ..................
        struct atm_dev  *dev;             /* device back pointer */
        struct atm_qos  qos;              /* QOS */
        unsigned long   tx_quota,rx_quota; /* buffer quotas */
        atomic_t        tx_inuse,rx_inuse; /* buffer space in use */
        void (*push)(struct atm_vcc *vcc,struct sk_buff *skb);
        void (*pop)(struct atm_vcc *vcc,struct sk_buff *skb);
        ...........................
        struct sk_buff_head recvq;        /* receive queue */
        struct atm_vcc   *prev, *next;
        .....................
};
```

The flags contain flags indicating the VC state. family is the address family, i.e. either PF_ATMPVC (for a PVC) or PF_ATMSVC (for a SVC). vpi and vci contain the connection identifier. In our case, the vpi is set to 0, as the ENI card does not support any other value; and the vci is set to the specified value for the physical VCI. The atm_qos structure is used to store information about the QoS parameters about the VCI. The elements of the atm_qos structure are;

```
struct atm_qos {
        struct atm_trafprm txtp;      /* parameters in TX direction */
        struct atm_trafprm rxtp;      /* parameters in RX direction */
        unsigned char aal;
};

where

struct atm_trafprm {
      unsigned char   traffic_class; /* traffic class (ATM_UBR, ...) */
      int             max_pcr;       /* maximum PCR in cells per sec */
```

```
        int             pcr;            /* desired PCR in cells per sec */
        int             min_pcr;        /* minimum PCR in cells per sec */
        int             max_cdv;        /* maximum CDV in microseconds */
        int             max_sdu;        /* maximum SDU in bytes */
};
```

The aal contains the information whether the VCI is AAL5 or AAL0. In our case, the physical VCI is always of type AAL5. The traffic_class and pcr in both the transmit and receive direction is set to the traffic class (ubr or cbr) and PCR specified when creating the VATM. tx_qouta and rx_qouta is maximum buffer space allocated for the VC at any point of time and is set to 1024*1024 bytes. tx_inuse and rx_inuse contain information about how much of the above quotas are in use respectively. The push function points to the actual function that needs to be called when any data is received on the VC. In our implementation, there are two receive functions; one (vdev_recv_atm) is used when the VATM is created on the ATM and the other (vdev_recv) when it is on the Ethernet.

The information about each of the layers in the protocol stack on a given virtual device is stored in the structure called layer. The fields of this structure are as follows:

```
struct layer
{
        struct vdevice *vdev;           /* virtual device */
        enum layer_type type;           /* layer type (e.g., ATM_LY) */
        struct layer *upper;            /* layer above (NULL if none) */
        struct layer *lower;            /* layer below (NULL if none) */
        struct layer_stats stats;       /* layer statistics */
        void *priv;                     /* pointer to private layer data */

        int (*init)(struct layer *layer);
        void (*destroy)(struct layer *layer);
        int (*open)(struct layer *layer, struct atm_vcc *vcc);
        void (*close)(struct layer *layer, struct atm_vcc *vcc);
        int (*ioctl)(struct layer *layer, unsigned int cmd, void *arg);

        /* ``send'' invoked from upper layer to send to this layer */
        int (*send)(struct layer * layer, struct sk_buff *skb);
        /* ``recv'' invoked from lower layer to send to this layer */
        int (*recv)(struct layer * layer, struct sk_buff *skb);
};
```

The vdev points to the virtual device of which it is a layer of and the layer_type specifies what type of layer it is. The valid values are:

```
enum layer_type {
        AAL_LAYER = 0,
        SAR_LAYER,
        DLC_LAYER,
        AAL_DLC_LAYER,
        QOS_LAYER
};
```

The upper and lower are pointers to the layer above and the layer below respectively. The struct layer_stats maintains the statistical information about each layer. The fields of this structure is as follows:

```
struct layer_stats
{
        unsigned long tx_packets;       /* total packets transmitted */
        unsigned long rx_packets;       /* total packets received */
        unsigned long rx_errors;        /* bad packets received */
        unsigned long tx_errors;        /* bad packets transmitted */
};
```

The init, open, destroy, close, ioctl, send and receive are pointers to service interface routines. For example, the open points to the actual open function (say sar_open) and is responsible for doing all that is necessary to create that layer. Similarly, the destroy and close point to functions that care of all the actions that is necessary when the layer is being destroyed. ioctl points to a function which takes care of changing any parameters on the fly. send and receive point to the actual send and receive functions of that layer (say, sar_send and sar_receive)

The data structure which stores information about the SAR layer is as given below:

```
struct sar_layer {
        struct layer * layer;
        struct sk_buff * vc_table[MAX_VCI];
                    /* VC descriptor table containing status info of each VC */
        char hec_table[MAX_VCI];
                    /* table to store pre-computed HEC fields for each open vc. */
};
```

where MAX_VCI (1024) is the maximum number of VCs that be opened simultaneously on the ATM card.

The data structure which stores information about the DLC layer is as given below:

```
struct dlc_layer {
        struct layer * layer;
        struct ahdlc_cb * cb;
};
```

The ahdlc_cb structure is used Adaptive HDLC and has not been implemented yet.

The data structure which stores information about the AAL_DLC_GLUE_LAYER layer is as given below:

```
struct aal_dlc_glue_layer {
        struct layer * layer;
        struct sk_buff * vc_table[MAX_VCI];
                  /* VC descriptor table containing status info of each VC */
        char hec_table[MAX_VCI];
};
```

3.1.3 VATM Commands[5]

The command to create the individual VATMs is provided as a patch to the ATM tools (ver 0.53). The syntax of the command to create the VATM is as follows:

vatm_ctl -c {type=atm} interface_id pvc=## {type=sar|dlc|sar+dlc|sar+qos+dlc} [local_esi_addr] [qos ubr/cbr:pcr=X]

- ***vatm_ctl -c*** option is to create the VATM
- ***{type=atm}*** specifies the fact that the VATM would be created over an ATM card
- ***interface_id*** indicates the interface # of the ATM card
- ***pvc=##*** specifies the physical VCI with which the VATM would be hooked to the ATM card.
- ***{type=sar|dlc|sar+dlc|sar+qos+dlc}*** specifies the protocol stack for the VATM
- ***[local_esi_addr]*** specifies the ESI (End System Identifier) address of the VATM
- ***[qos ubr/cbr:pcr=X]*** specifies the qos parameter for the physical VCI

E.g. *vatm_ctl –c atm 0 pvc=201 sar+dlc 0020ea000130 qos cbr:pcr=10Mbps*

This would create a VATM which would have SAR+DLC in its protocol stack, and would be hooked to the ATM card on VCI=201 with its PCR=10Mbps. This can be verified by checking the /proc/atm/devices; which would look as shown below:

```
testbed11 [2] % cat /proc/atm/devices
Itf Type    ESI/"MAC"addr AAL(TX,err,RX,err,drop) ...
  0 eni     0020ea002d24  0 ( 0 0 0 0 0 )  5 ( 0 0 0 0 0 )
  1 vatm    0020ea000130  0 ( 0 0 0 0 0 )  5 ( 0 0 0  0 0 )
```

[5] Please check Appendix A.1 for the complete set of commands related to VATM.

3.2 WMRP (Wireless Multi-Path Routing Protocol)

3.2.1 Notation and Assumptions[5]

To describe WMRP, we model the network as an undirected graph represented as G(V,E), where V is the set of nodes and E is the set of links (or edges) connecting the nodes. Each node represents a router, where (u,v) is an edge of the graph G, u is said to be adjacent to v (or a neighbor of v), and v is a neighbor of u. A route from node x to destination node j is a sequence of adjacent nodes $(x, k_1, k_2, \ldots k_n, j)$ denoted by R_{xj}. A path from x to j via node k is denoted R^{k}_{xj} where k is a node adjacent to x. The distance is the number of hops between x and j (sum of link weights each of weight equal to 1) $D(R_{xj})$. The predecessor node of a path R_{xj} is defined to be the last node preceding node j in the sequence of node in R_{xj}. denoted as $P(R_{xj})$. N_x is a list (set) of neighbors of node x.

The description makes the following assumptions:

- Each node has a unique identifier.

  From the implementation perspective, this is the virtual IP address assigned to the IP over ATM interface.
- A node detects the existence of a new neighbor within a finite time (order wire agent).
- An underlying (Data Link Control) protocol ensures that packets are delivered correctly. (Reliable transmission of update messages can be implemented by means of retransmissions if no such layer exists). From the implementation perspective, this implies that TCP should be used to exchange the Hello Packets and the routing table.

All links are assigned the same weight (a value of 1). All links between MAP nodes in RDRN are identical with respect to bandwidth and delay. Reliability (error rate), on the other hand, is highly variant during the operational period of the link; so it would require a constant monitoring and a change in the link weight if the reliability is to be included in the calculation of the link weights. This would increase the routing overhead.

3.2.2 Information maintained at each node[5]

Each node maintains the following information

1. A *routing table,*
2. A list of neighbors (N_x) and
3. A list of new neighbors (B_x).

The routing table of node x contains an entry for each destination j, each of those entries contain the following:

- The destination identifier (j).
- A list of neighbors, if any, that would be affected by the change and need to be informed about it (N_{xj}).
- One or more path(s) information, each path is identified by a different neighbor and contains the following information:
  - The identifier of the next hop (k).
  - Metric representing the distance to j $D(R^k{}_{xj})$.
  - The identifier of the predecessor (next to last) $P(R^k{}_{xj})$.
  - Beam number assigned for the next hop

It should be noted that the number of paths from node x to any destination is limited by the degree (number of adjacent nodes) of node x because each entry (path) is identified by a different neighbor. When the routing agent is informed about a new neighbor k (through order wire agent) and that a new link has been established, x adds k to the list of new neighbors (B_x), x and k start exchanging Hello messages. Three Hello messages have to be exchanged within three Hello Intervals (Hold-off time) before the two nodes exchange their routing information. This procedure is adopted for the following reasons:

1. It allows the nodes to check wireless reachability (not just order wire reachability) and makes sure that the link is reliable.
2. It checks that the other node is not just passing by quickly.
3. It gives the nodes enough time to detect any link failures with previous neighbors. A new link coming up, would be due to a change in the topology, which means the one of the two nodes (if not both) has moved and most probably lost connectivity with one or more of its previous neighbors. The hold-off would give enough time for invalid routes to timeout before exchanging information. This prevents incorrect information from being propagated through the network.

Note that four Hello messages can be exchanged with a three Hello interval period, so this works as a 3 out-of 4 system. If three Hellos are exchanged successfully within the hold-off time (three Hello intervals), then the two nodes will exchange their routing information (x will remove k from B_x and add it to N_x, and send it a summary of its routing table). Otherwise, x removes k from its list of new neighbors (B_x).

3.2.3 Information exchanged among nodes[5]

Nodes exchange periodic Hello Packets. Routing table update messages are only sent after a change in the routings table (reflecting a change in the topology). Updates are attached with the next Hello Packet. Each entry ($U^k{}_j$) in an update packet (U^k) from a neighbor k about destination j contains the following:

- The identifier of the destination node *j*.
- The distance to j $D(U^k{}_j)$.
- The identifier of the predecessor node $P(U^k{}_j)$.

3.2.4 Processing an Update[5]

When node *x* receives an update message for destination j (route R_{xj}) from a neighbor k, it verifies the information in the update by checking if it has in its routing table an entry for the predecessor with the same neighbor k ($R^k{}_{xp}$) and a distance $D(U^k{}_j)$. If the update is verified, x updates its table and set the distance to destination j ($D(R^k{}_{xj}) = D(U^k{}_j) +1$); otherwise $D(R^k{}_{xj})$ is set to ∞. The Update procedure checks the neighbors that will be affected by the change in the route $R^k{}_{xj}$ and adds those neighbors to N_{xj}. Appendix B.2 shows the update algorithm in more details.

When an update is made to an entry in *x*'s table (R_{xj}), the neighbors that are affected by this change are added to the neighbors' list (N_{xj}) so that they can be informed about the change with the next Hello packet. Note also that *x* informs each neighbor k about the minimum path it has to a destination j that doesn't have k as the next hop or predecessor (split horizon); so x will send an update k only if that minimum changes.

3.2.5 Sending Updates[5]

The Send-Update procedure (shown in Appendix B.3) is called every Hello interval to send a Hello packet to each neighbor, and checks which updates should be sent to which neighbors. Send-Summary procedure, which is responsible for the exchange of Hello Packets (shown in Appendix B.3) is called after successfully exchanging three hello packets with a neighbor. The complete algorithm is shown in the Appendix B.1.

3.2.6 Implementation of WMRP

The routing protocol was implemented in the user-space and Netlink sockets were used to change the kernel routing table. Since, the routing protocol requires different "events" (like sending the initial 3 Hello Packets, checking for the change in the topology, etc.) to be executed concurrently; it had to be implemented either as multithreaded or as a multi-process application. It was implemented as multi-threaded program using Pthreads[6]. The reasons why we chose to implement it as a multi-threaded program as compared to a multi-process program are[9]:

- Creating a process can be expensive, both in time and memory
- Synchronization of data is easier with Pthreads.

Besides the routing table the routing module maintains two lists:

- `new_neighbor_list`:

  This is the list of nodes to which the routing module tries to exchange the 3 Hello Packets within the Hello Interval
- `neighbor_list`:

  This is the list of nodes which are the neighbors and to which the high-speed connectivity exists.

3.2.6.1 Interaction between the Orderwire Module and the Routing Module

[6] Pthreads is a standardized model for dividing a program into subtasks whose execution can be interleaved or run in parallel. The "P" in Pthreads comes from POSIX[9].

The orderwire module writes to the shared memory about all the nodes to which it has high-speed connectivity and on what beam # as shown in Figure 22.

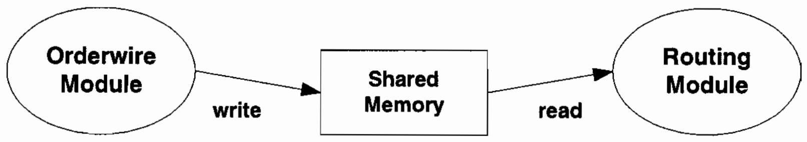


Figure 22: Interaction between Orderwire Module and Routing Module

The routing module reads from the shared memory and checks whether the node is already included in the `neighbor_list` or not. If not, it would add that node to the list of the `new_neighbor_list` and start the exchange of the initial 3 Hello Packets.

3.2.6.2 Threads in the Routing Module

The different threads that constitute the routing module are shown in Figure 23.

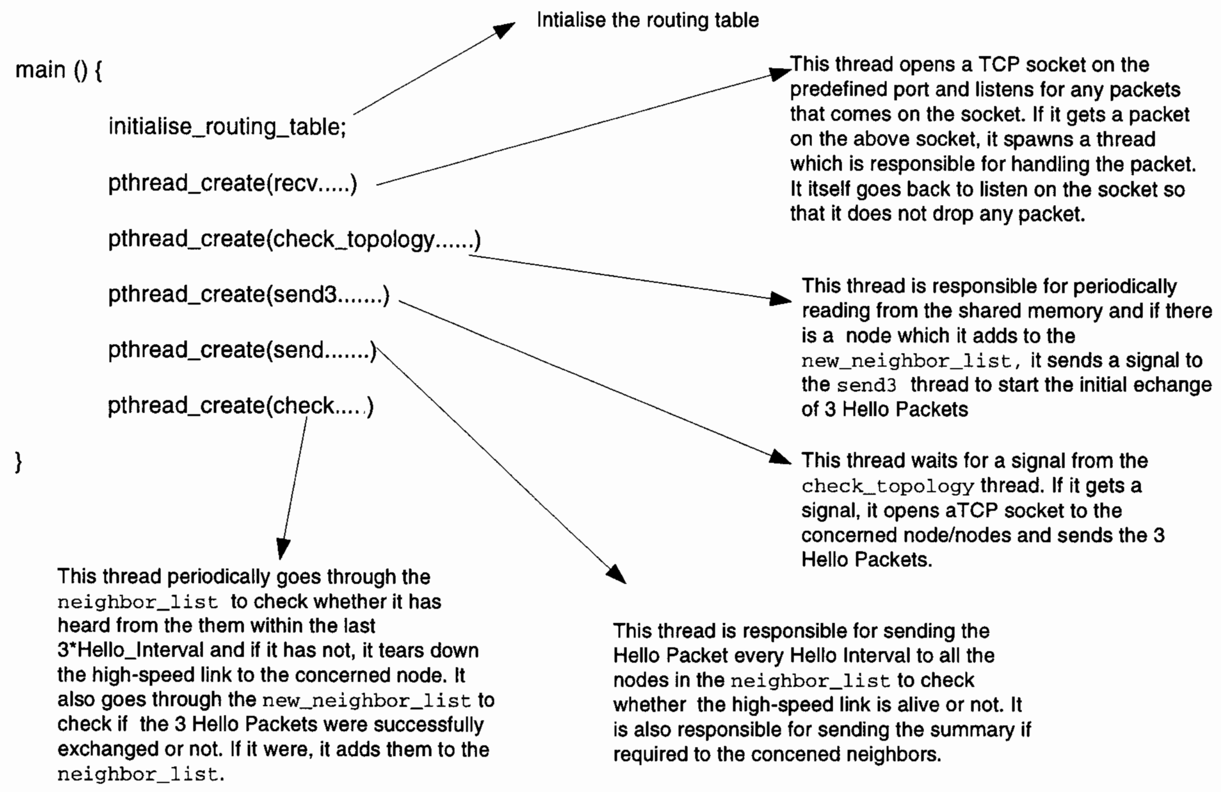

Intialise the routing table
main () {
initialise_routing_table;
pthread_create(recv.....)
pthread_create(check_topology......)
pthread_create(send3.......)
pthread_create(send.......)
pthread_create(check.....)
}
This thread opens a TCP socket on the predefined port and listens for any packets that comes on the socket. If it gets a packet on the above socket, it spawns a thread which is responsible for handling the packet. It itself goes back to listen on the socket so that it does not drop any packet.
This thread is responsible for periodically reading from the shared memory and if there is a node which it adds to the new_neighbor_list, it sends a signal to the send3 thread to start the initial echange of 3 Hello Packets
This thread waits for a signal from the check_topology thread. If it gets a signal, it opens aTCP socket to the concerned node/nodes and sends the 3 Hello Packets.
This thread periodically goes through the neighbor_list to check whether it has heard from the them within the last 3*Hello_Interval and if it has not, it tears down the high-speed link to the concerned node. It also goes through the new_neighbor_list to check if the 3 Hello Packets were successfully exchanged or not. If it were, it adds them to the neighbor_list.
This thread is responsible for sending the Hello Packet every Hello Interval to all the nodes in the neighbor_list to check whether the high-speed link is alive or not. It is also responsible for sending the summary if required to the concened neighbors.

Figure 23: Threads in the Routing Module

Since, more than one thread would attempt to read/write to the same variable, synchronization of the threads was required. This was done using the mutex variable functions. A mutex variable acts as a mutually exclusive lock, allowing threads to control the access to data. The threads agree that only one thread at a time can hold the lock and access the data it protects.

3.2.6.3 Changing the Kernel Routing Table

As outlined earlier, the implementation of the WMRP resides in the user-space. However, the routing table in the kernel needs to be changed to reflect the current routing table that resides in the user-space. Hence, as when the shortest path to the concerned destination changes, the routing module uses Netlink sockets to change the kernel routing table.

Netlink is used to transfer information between kernel modules and user space processes. It provides kernel/user-space bi-directional communication links. It consists of a standard sockets based interface for user processes and an internal kernel API for kernel modules. Netlink is a datagram-oriented service. The interface is as shown below:

```
netlink_socket=socket(PF_NETLINK,socket_type,netlink_family);
```

Both SOCK_RAW and SOCK_DGRAM are valid values for `socket_type`; however the netlink protocol does not distinguish between datagram and raw sockets. `netlink_family` selects the kernel module or netlink group to communicate with. The `netlink_family` used in our case was:

- NETLINK_ROUTE: Receives routing updates and is used to modify the IPv4 routing table.

It is the mark of an educated mind to rest satisfied with the degree of precision which the nature of the subject admits and not to seek exactness where only an approximation is possible.

Aristotle

Chapter 4

Results

In this chapter, two scenarios were executed in the emulation environment and metrics like throughput, time for the network to converge & fraction of the emulation time for which connectivity could be maintained were measured.

4.1 Scenario 1

The following scenario (Figure 24) consisting of 7 nodes were tested. It consisted of 4 MAPs and 3 MEPs.

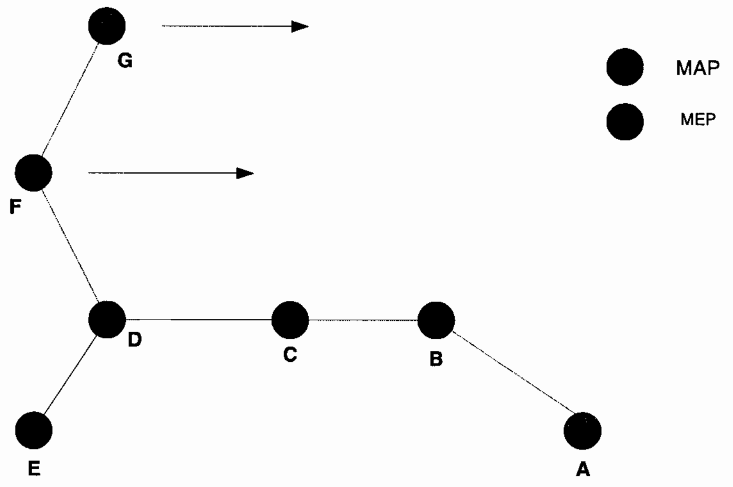


Figure 24: Scenario 1

In the above scenario all the nodes except Node F & Node G were stationary. Node F moves eastward in a straight path at a speed of 436.98 Km/hr & Node G also moves eastwards in a straight path at a speed of 460 km/hr. The speed of the nodes was chosen to be high so that it is possible to see appreciable change in the topology for smaller emulation periods. Please look at the Appendix D for the scenario file. The different states of connectivity of the network is as shown below:

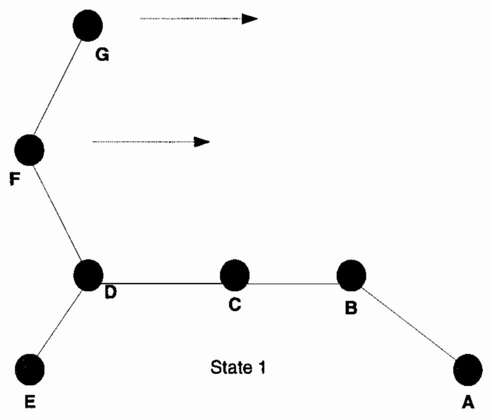


Figure 25: State 1 of Connectivity Scenario 1

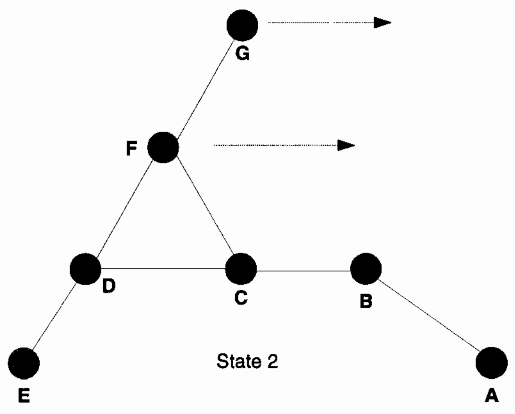


Figure 26: State 2 of Connectivity in Scenario 1

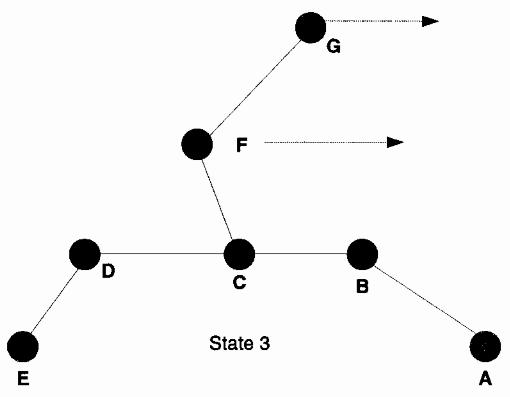


Figure 27: State 3 of Connectivity in Scenario 1

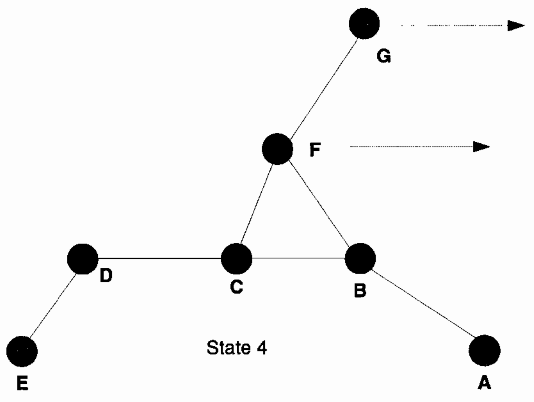


Figure 28: State 4 of Connectivity in Scenario 1

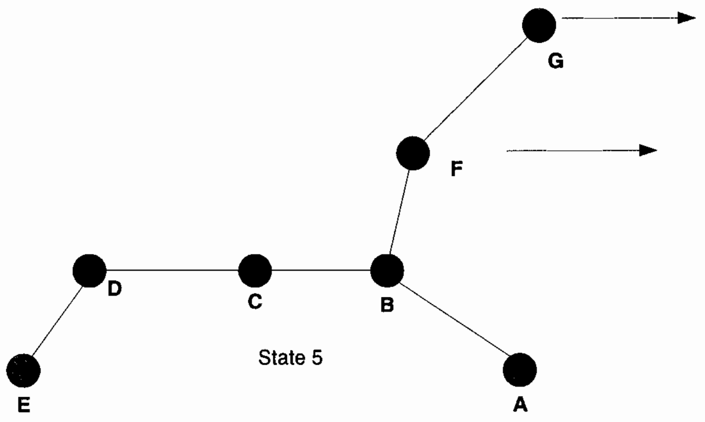


Figure 29: State 5 of Connectivity in Scenario 1

The emulation starts i.e. the Emulation Manager started to transmit the GPS data to each of the nodes, when all the 7 nodes in the scenario had registered with the Emulation Manager. This way it was ensured that all the nodes came up at the same time. Moreover, the clocks of the testbeds (the nodes) were synchronized with that of the Emulation Manager.

4.1.1 Ping Results

The average RTT between Node A & Node G for the ping application was measured by forcing 2000 ping packets. This was done whenever it was observed that the route between Node A & Node G had changed. The graph below (Figure 28) shows the result obtained.

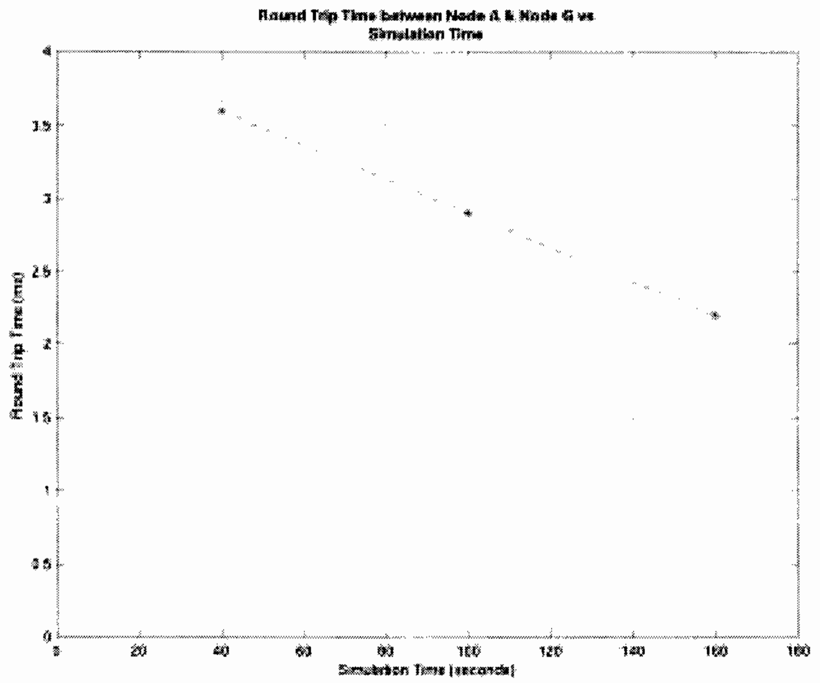


Figure 30: Round Trip Time (RTT) for Ping vs Emulation Time

As it can be seen from the above graph that the RTT time decreased as the simulation time increased. This is so because the path taken by the packets changed as follows:

From {A→B→C→D→F→G} to {A→B→C→ F→G} to {A→B→F→G}.

4.1.2 Connectivity Results

The following graph (Figure 31) indicates the time taken to establish connectivity and also the state of connectivity between Node A & Node G for different Hello Intervals of the routing protocol.

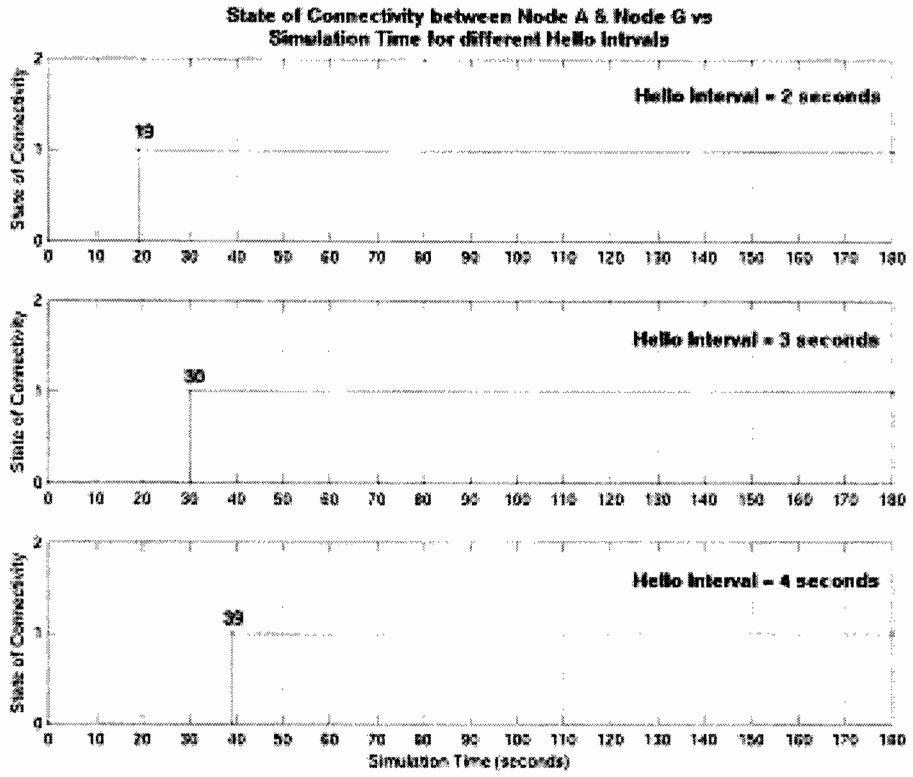


Figure 31: State of Connectivity between Node A & Node G for different Hello Intervals

It should be noted here that the time taken to establish connectivity between Node A & Node G has been measured with respect to the time at which the emulation was started i.e. at time t = 0, even Node F did not know about the existence of Node G.

As expected, the time taken to establish connectivity between Node A & Node G increases with the increase in the Hello Interval. The smaller the Hello Interval, the faster the nodes would exchange their routing table and hence the routes to different destinations would propagate faster across the network.

It can be seen from Figure 31 that once the connectivity between Node A & Node G has been established, it does not go down. This can be explained by looking at the state of connectivity diagrams (Figure 25- Figure 29). First, the connectivity between Node A & Node G is via A↔B↔C↔D↔F↔G as shown in Figure 25. When Node F comes within the range of Node C, the link between Node F & Node C is established, and hence the connectivity between Node A & Node G is via A↔B↔C↔F↔G i.e. Node C would route packets for Node G through Node F and *not* through Node D. Hence, even when the Node F-Node D link goes down (Figure 27) the connectivity between Node A & Node G is maintained. It can be similarly explained how the connectivity between Node A & Node G is maintained even when the Node C-Node F link goes down.

The following graph (Figure 32) indicates the time taken to establish connectivity and also the state of connectivity between Node E & Node G for different Hello Intervals.

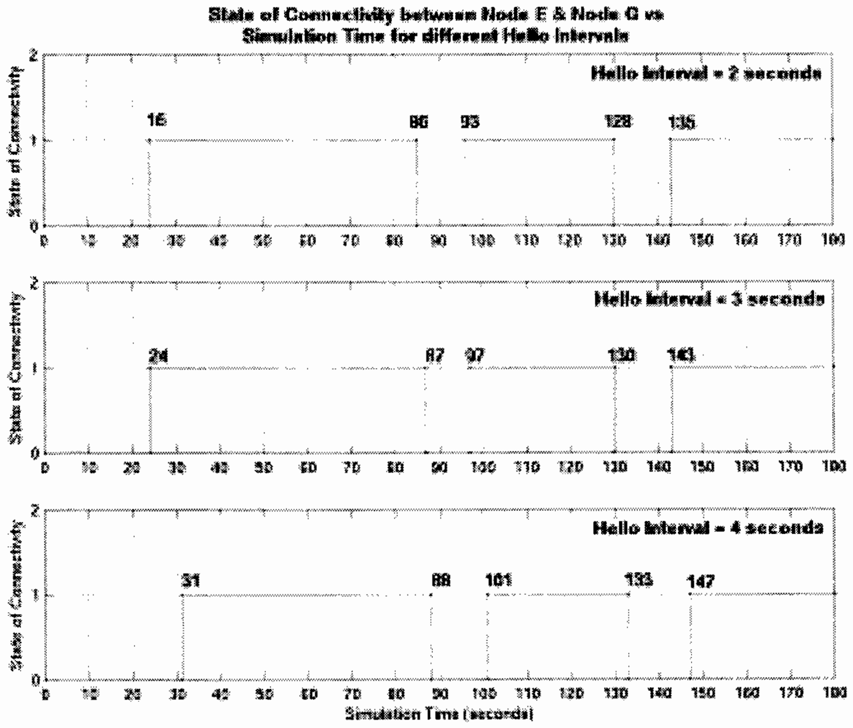


Figure 32: State of Connectivity between Node E & Node G for different Hello Intervals

It should be noted here that the time taken to establish connectivity between Node E & Node G has been measured with respect to the time at which the simulation was started i.e. at time t = 0, even Node F did not know about the existence of Node G.

As expected, the time taken to establish connectivity between Node E & Node G increases with the increase in the Hello Interval. The smaller the Hello Interval, the faster the nodes would exchange their routing table and hence the routes to different destinations would propagate faster across the network.

However, if we compare the time taken to establish the connectivity between Node E & Node G and Node A & Node G for the same Hello Interval, we can see that the time taken for the Node E-Node G pair is less. This is so because the initial shortest path between Node E & Node G was across only 3 hops while it was 5 hops in the case of Node A-Node G pair.

We can see from Figure 30 that the connectivity between Node E & Node G is not maintained at all times. This can be explained by looking at the state of the connectivity diagrams (Figure 25-Figure 29). Initially, the connectivity between Node E & Node G is via E↔D↔F↔G (Figure 25). Even, when the Node F-Node C is established, Node D would still route the packets for Node G through Node F only as it would be the shortest path and hence the above route would be maintained. However, when the link between Node D & Node F goes down, the connectivity is lost and it would take the routing protocol at Node D at least 3*Hello_Interval to detect the link failure. When it is able to do so, it would start routing the packets for Node G through Node C. This is why we see the loss of connectivity at t=86 (for Hello Interval = 2seconds). However, we can see that the connectivity is up again after 7 seconds (3*Hello_Interval +1) though it should have 6 seconds for the Hello Interval = 2 seconds. The extra second can be attributed to the fact the resolution of the tool, which checked for the connectivity, was 1 second.

4.1.3 Throughput Results

The throughput measurements were made between Node A and all the nodes in the network for Hello Interval = 2 seconds. They were measured by using the FTP application and the size of the file transferred was approximately 20 MB. The FTP application was started as soon as the connectivity between Node A and the other given node was achieved. The measurements were made for 3 different cases; in each case, the

VATM was configured with a different protocol stack. The different combinations of the protocol stack were:

- SAR +DLC

  In this case the IP packet coming from the higher layer (as CLIP) was segmented into a train of ATM cells and passed to the DLC. The number of cells in the DLC packet was 7. The DLC packet was passed to the ATM card which sent it as an AAL5 packet.

- SAR

  In this case the IP packet coming from the higher layer (as CLIP) was segmented into a train of ATM cells which passed to the ATM card. The ATM card sent the train of cells as an AAL5 packet.

- DLC

  In this case the IP packet coming from the higher layer (as CLIP) was put in a DLC packet and passed to the ATM card which sent it as an AAL5 packet.

In each of the 3 cases, *no* software ATM switching was done at the MAPs; instead IP forwarding was enabled on all the MAPs. The throughput observed has been shown in Figure 33 and Table 3.

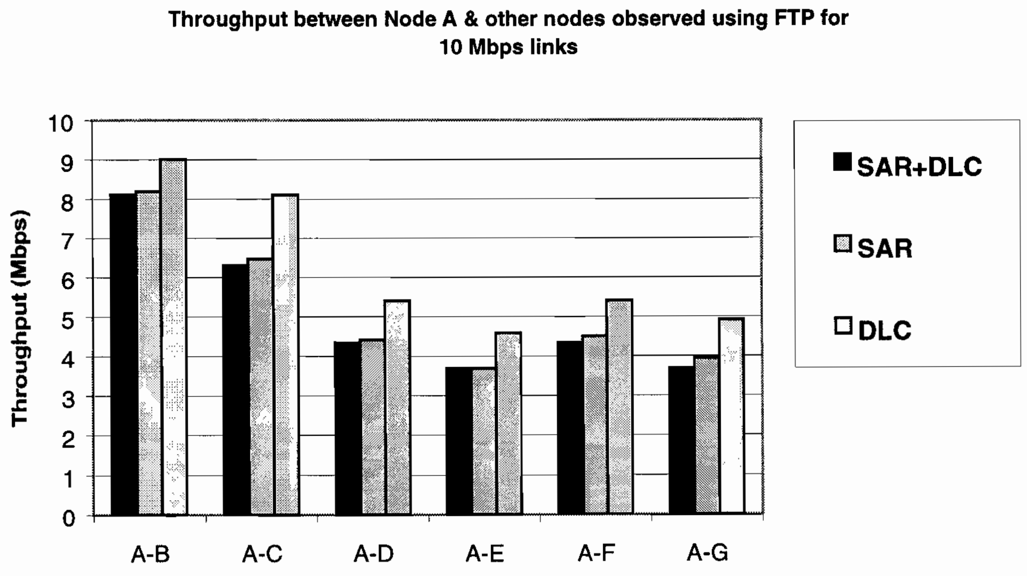


Figure 33: Throughput Results for the 7 Node Scenario with different protocol stacks on the VATM

Source-Destination Pair	**SAR+DLC (Mbps)**	**SAR (Mbps)**	**DLC (Mbps)**
A-B	8.1108	8.192	9.0122
A-C	6.30784	6.4717	8.1108
A-D	4.3418	4.4237	5.4067
A-E	3.6864	3.6864	4.5875
A-F	4.3418	4.5056	5.4067
A-G	3.6864	3.9322	4.9152

Table 3: Throughput Results for Different Protocol Stack on the VATM

The features of the results that should be noted are:

- Minimum throughput was observed (in all the 3 cases) between Node A & Node E. This was expected as the number of hops between Node A & Node E was 4 at all instances.
- Throughput between Node A & Node G was more than between Node A & Node E even though initially the route between Node A & Node G was 5 hops. This is so because as Node F & Node G moved eastwards, the route taken by packets from Node A to Node G changed from 5 hops to 4 hops and finally to 3 hops.
- The least throughput was observed when the VATM protocol stack was configured as SAR+DLC. This is so because of the overhead of the SAR layer which would have attached 5 bytes for every 48 bytes and also because of the DLC layer which would have added 8 bytes (header + trailer) for every 7 ATM cells.
- The throughput when the VATM protocol stack was configured as SAR only was greater than the SAR+DLC case but less than that observed for the DLC only case. The reason for the former is that the DLC overhead i.e. 4 bytes of header and 4 bytes of trailer, for every 7 ATM cells were not there.
- The throughput was highest when the VATM protocol stack was configured as DLC only. This is so because there was no SAR overhead and the DLC

overhead were also reduced. In case of SAR+DLC, there were 8 bytes of DLC header for every 7 ATM cells. This was reduced to 5 bytes (AAL_DLC_GLUE_LAYER) plus 8 bytes (DLC) for every IP packet.

The throughput between Node A & Node G was also measured when the VATM was configured with the SAR+QoS+DLC protocol stack. In this case, software ATM switching was done at the MAPs unlike IP forwarding in the above 3 cases. The support for the flow establishment/re-establishment was implemented by Saravanan Radhakrishnan. More detailed information about it can be found in his thesis titled "In-Band Flow Establishment for end-to-end QoS Support in RDRN". In this case the tool used to measure the throughput was a modified (by Saravanan Radhakrishnan) version of *ttcp*. The results for different size of packets has been shown in Table 4:

# of Packets	**Size of the Packets** (Bytes)	**Tx Rate** (Mbps)	**Rx Rate** (Mbps)
2048	512	3.6618	3.6563
2048	1054	6.7584	6.5782
2048	1536	9.8877	9.6583

Table 4: Throughput Result for SAR+QoS+DLC stack on the VATM

Hence, if we compare the results from Table 3 & Table 4, we can see that for small packets the throughput achieved for the SAR+QoS+DLC (i.e. ATM connectivity) is actually less than that observed with IP connectivity. However, the throughput achieved for larger packets is appreciably greater than that observed with IP connectivity.
The throughput achieved between Node A & Node G when they had IP connectivity measured with *ttcp* is as follows for the SAR+DLC protocol stack:

# of Packets	**Size of the Packets** (Bytes)	**Throughput** (Mbps)
2048	512	3.9813
2048	1054	4.4237
2048	1536	5.5296

Table 5: Throughput between Node A & Node G for IP connectivity with SAR+DLC

4.2 Scenario 2

The following scenario consisted of 7 nodes, 5 MAPs and 2 MEPs as shown in Figure 34.

All the nodes are moving except for Node G. The MAPs traverse along the perimeter of the exterior pentagon while Node F traverses along the perimeter of the interior pentagon. At the end of the emulation period, they return back to the GPS location they started from. The speed at which the nodes move is 480 km/hr. The different states of connectivity have been shown in Figure 35 & Figure 39.

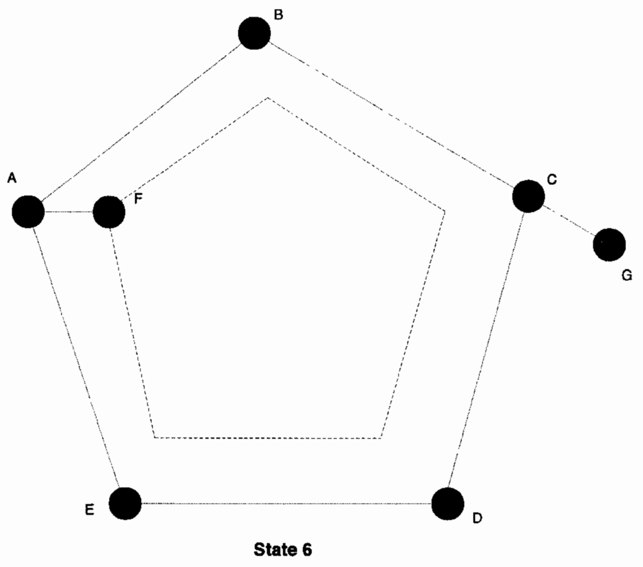


Figure 34: Scenario 2

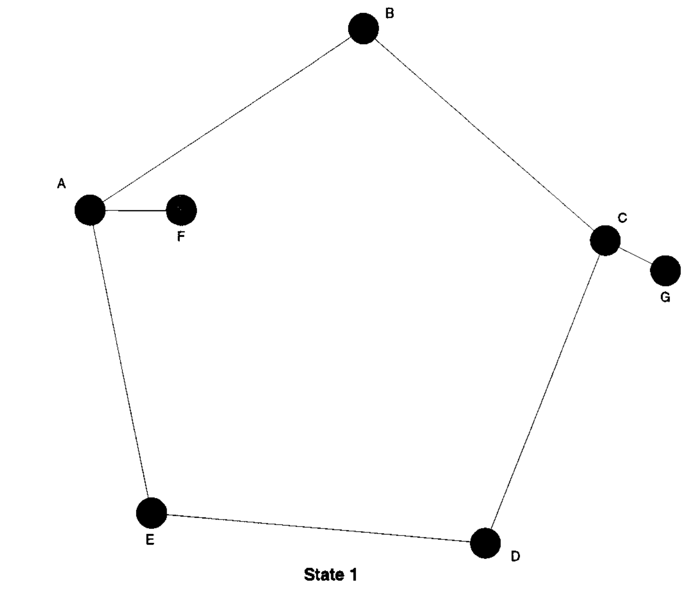


Figure 35: State 1 of Connectivity in Scenario 2

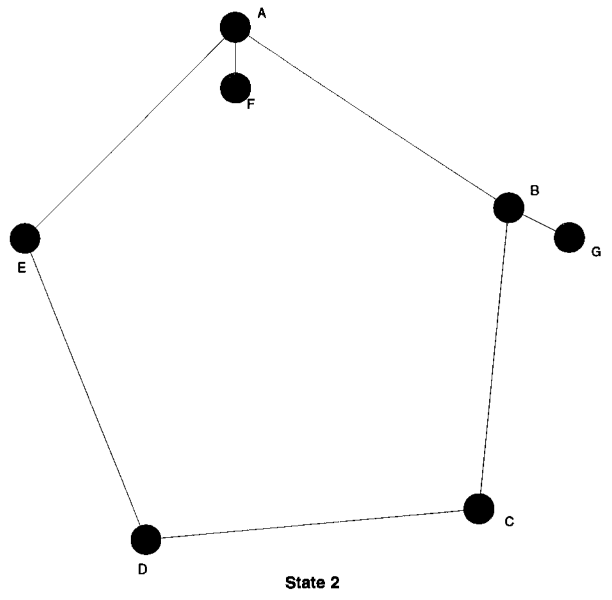


Figure 36: State 2 of Connectivity in Scenario 2

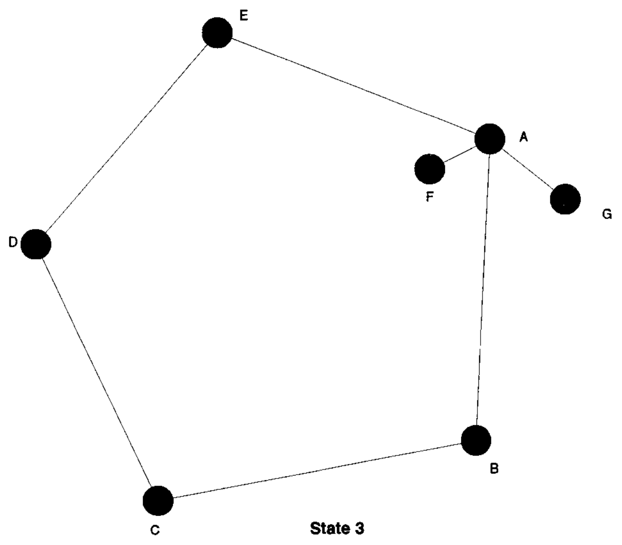


Figure 37: State 3 of Connectivity in Scenario 2

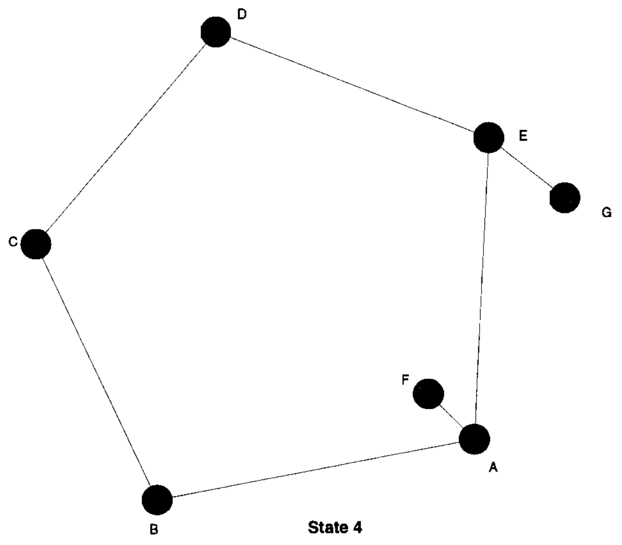


Figure 38: State 4 of Connectivity in Scenario 2

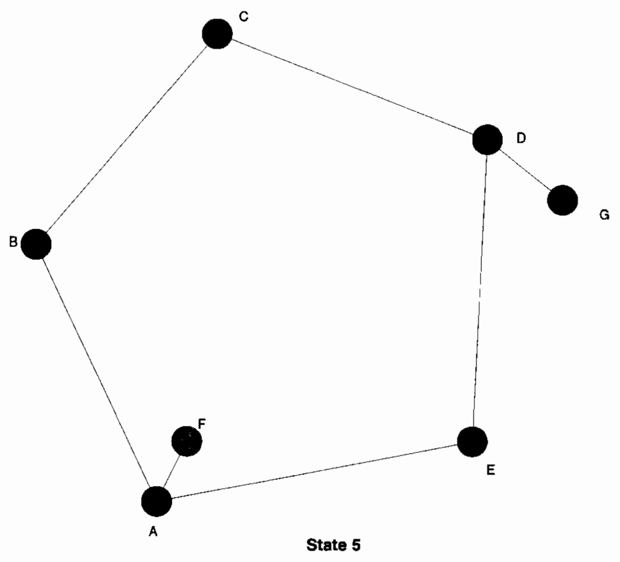


Figure 39: State 5 of Connectivity in Scenario 2

4.2.1 Ping Results

The average RTT between Node F & Node G for the ping application was measured by forcing 2000 ping packets. This was done whenever it was observed that the route between Node F & Node G had changed. The graph below (Figure 40) shows the result obtained.

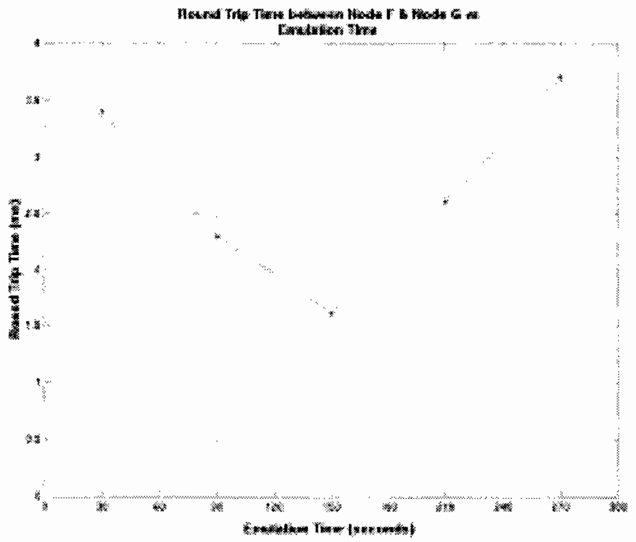


Figure 40: Round Trip (for ping) between Node F & Node G vs Emulation Time

The state of connectivity of the network at t=30s, 90s, 150s, 210s 270s was as shown in Figure 35-39.

Hence, the route taken by a packet from Node F to Node G at the above time was:

{F→A→B→C→G} {F→A→B→G} {F→A→G} {F→A→E→G} {F→A→E→D→G}

4.2.2 Connectivity Results

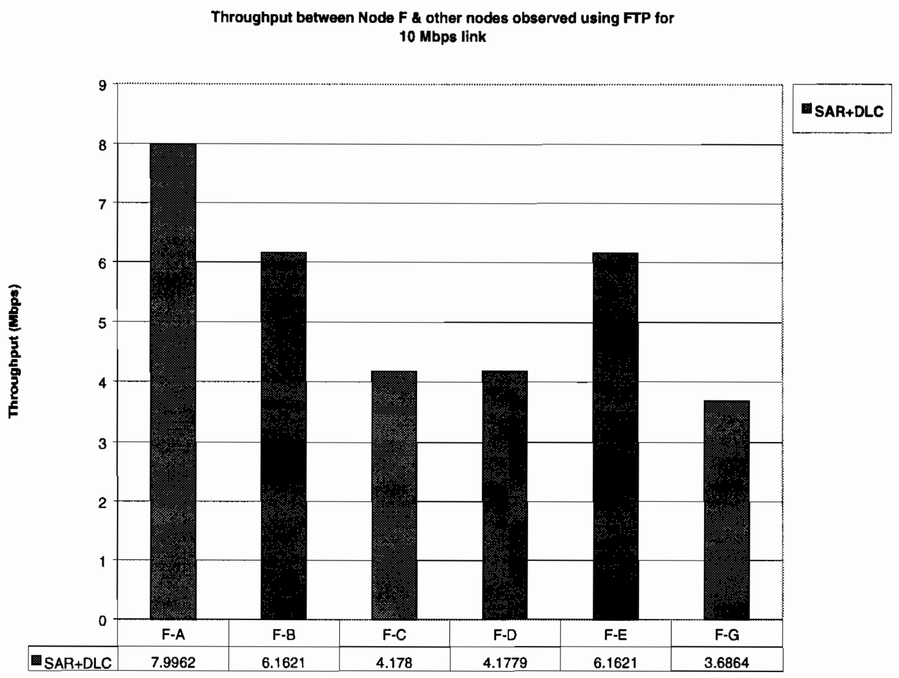
The emulation time for this scenario was 1 hour and the time taken to establish connectivity between Node F & Node G was measured for different Hello Intervals. The results are as shown in Table 5. As expected the time taken to establish the connectivity was minimum for the smallest Hello Interval (2 seconds).

Hello Interval (seconds)	**Time (seconds) taken to Establish Connectivity**
2	18
3	24
4	27

Table 6: Time taken to establish connectivity between Node F & Node G

4.2.3 Throughput Results

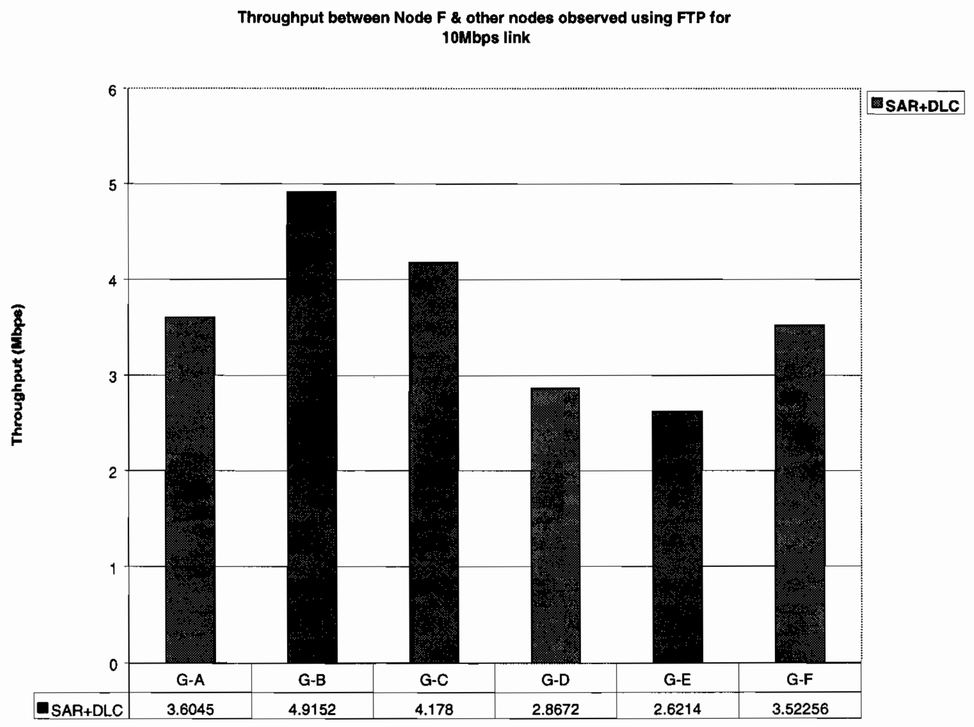
For this scenario, the throughput measurements were made only for the SAR+DLC case only. The following graph shows the throughput between Node F and the other nodes for 10 Mbps link using the FTP application. The size of the file transferred was approximately 20Mbps. The FTP application was started as soon as connectivity was established between Node F and the other concerned node. The result observed has been shown in Figure 41.



	F-A	F-B	F-C	F-D	F-E	F-G
■ SAR+DLC	7.9962	6.1621	4.178	4.1779	6.1621	3.6864

Figure 41: Throughput Results between Node F and other nodes in Scenario 2 with SAR+DLC VATM

The following graph shows the throughput between Node G and the other nodes for 10 Mbps link using the FTP application. The size of the file transferred was approximately 20Mbps. The FTP application was started as soon as connectivity was established between Node G and the other concerned node. The result observed has been shown in Figure 42.



	G-A	G-B	G-C	G-D	G-E	G-F
■ SAR+DLC	3.6045	4.9152	4.178	2.8672	2.6214	3.52256

Figure 42: Throughput Results between Node G and other nodes in Scenario 2 with SAR+DLC VATM

We can see from the above graph (Figure 42) that the throughput between G-B is greater than that observed between G-C, though Node C was a neighbor of Node C to start with. The FTP application was started as soon as connectivity was achieved between Node G and Node C, but however the file transfer could not complete before the Node G-Node C link went down. Hence, some packets from Node G to Node C took the G-C path and some took the G-B-C path; with more taking the latter path. However, for the file transfer between Node G and Node B, fewer packets took the G-C-B path and more took the direct G-B path.

This shows that the above throughput results are a not a very accurate indicator of connectivity for this particular scenario because the throughput observed was dependent at what time the FTP application was started.

If you follow reason far enough it always leads to conclusions that are contrary to reason.

Samuel Butler

Chapter 5

Conclusions & Future Work

This chapter briefly draws out the conclusion of this work and looks in to possible areas of work that need to be addressed in the future.

5.1 Conclusion

The three major objective of this work was to do a design and implement an emulation environment, do a comparative evaluation of IP vs ATM for a highly dynamic environment like RDRN and to measure the scalability of the network controller (software modules). The conclusion for each of the above has been outlined in the following section.

5.1.1 Emulation Environment

We successfully implemented a repeatable, a controlled and a scalable emulation environment with minimal changes to the existing software modules.

5.2.1 IP vs ATM

We saw in Section 4.1.3 that the maximum throughput was observed between any two nodes when the VATM was configured as SAR+QoS+DLC. This was followed by the DLC case and then by the SAR+DLC case. This showed that the maximum throughput could be achieved if the end-to-end connectivity between the RDRN nodes was ATM based and a QoS layer was there to do the scheduling. However, the problem with this approach is that to utilize the QoS layer, the source-code of the applications need to be modified to set the options on the socket and to set the priority for the traffic.

The throughput observed when the RDRN nodes had end-to-end IP connectivity (SAR+DLC and DLC case) was obviously less than that for the ATM connectivity because each of the IP packets were reassembled at the intermediate nodes before being forwarded depending upon the entries of the routing table. However, the throughput observed for the DLC protocol stack on the VATM was greater than that for the SAR+DLC protocol stack. This was because the overhead of segmentation and re-assembly was removed and the DLC overhead per IP packet was also reduced.

5.2.2 Scalability

Before this work, the software modules (network controller) had been used only for a 3-node scenario. As part of this work, we executed two network scenarios, each consisting of 7 nodes, in the emulation environment and successfully demonstrated that software modules do scale-up. However, larger scenarios need to be tested in the emulation environment to get a more accurate measure of the scalability of the system.

5.2 Future Work

5.2.1 Topology Algorithm

The topology algorithm needs to be improved so as to guaranty connectivity to all the nodes in a complex network scenario. As explained earlier, on the MAPs, the topology algorithm tries to establish high-speed connection between the nearest four nodes and on the MEP it tries to establish the high-speed connection to the nearest MAP. Let us consider the following network scenario:

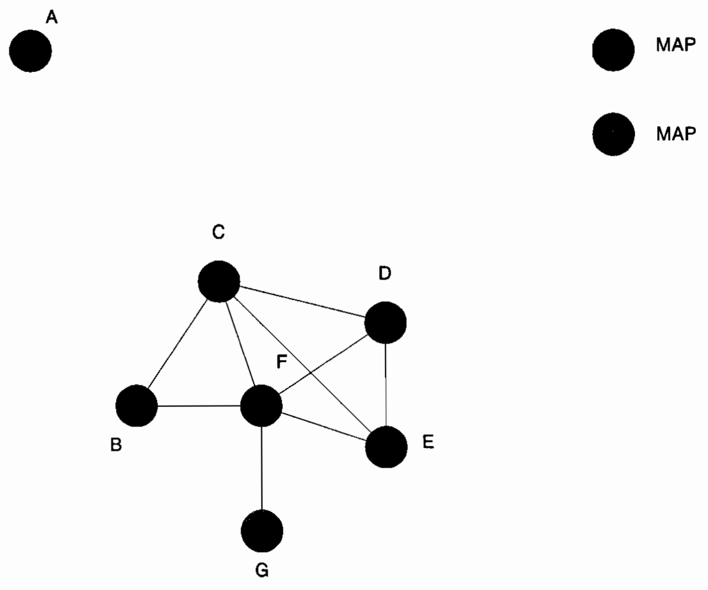


Figure 43: Network Topology with the present algorithm

Let us assume that Node A is within the high-speed range of Node C only. With the present topology algorithm, Node C would try to connect to the nearest 4 and hence would not connect to Node A i.e. Node C would connect to B, D, F & G. This way Node A would never get connected even though it was within the high-speed range of Node C.

Hence, more intelligence need to be added to the topology algorithm so that it could set up the following scenario:

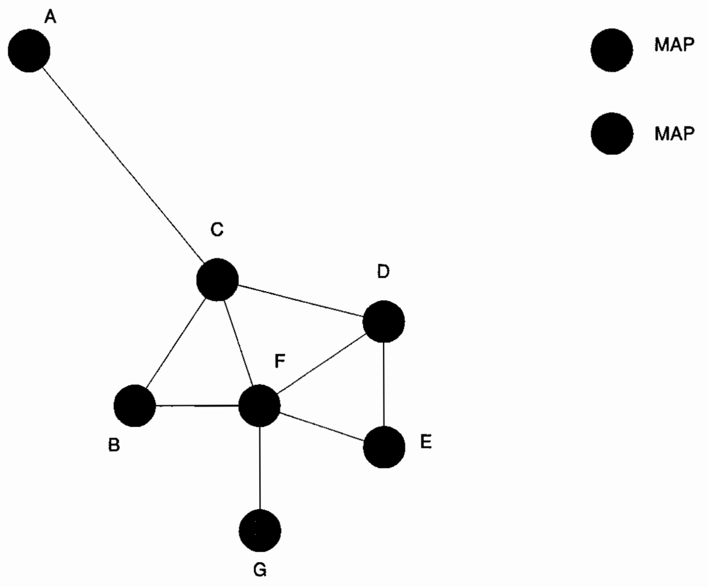


Figure 44: Network Topology with a more "intelligent" algorithm

In the above topology, Node C did not connect to Node E because it knew that if it did so it would be "isolating" Node A from the network.

5.2.2 Wireless Channel Model

The present emulation environment does not include any model to emulate the wireless channel characteristics. Hence, a possible area of future work would be include a layer in the VATM protocol stack which would incorporate the characteristics of the wireless channel; and a "handle" to control these characteristics at run-time.

5.2.3 Performance Metrics for Larger Scenarios

As part of this work, two network scenarios were executed in the emulation environment. Larger and more complicated network scenarios need to be tested, especially with the terrain blocking.

Appendix

Appendix A:

A.1 Commands Related to VATM:

i. ***vatm_ctl -l***

-- to list virtual devices

This command lists all the VATMs that have been created and lists the following information about the VATM:

```
-------------------------------------------------------
      VIRTUAL DEVICES & THEIR CONFIGURATION
-------------------------------------------------------
Itf = 1
DEVICE TYPE = ATM
VCI HOOK = 201
VPI HOOK = 0
My MAC ADDRESS = 0 20 ea 0 1 41
Underlying Interface Itf = 0
TOP LAYER
Layer AAL
       tx_packets = 0
       rx_packets = 0
       tx_errors = 0
       rx_errors = 0
INTERMEDIATE LAYER
Layer SAR
       tx_packets = 0
       rx_packets = 0
       tx_errors = 0
       rx_errors = 0
BOTTOM LAYER
Layer DLC
       tx_packets = 0
       rx_packets = 0
       tx_errors = 0
       rx_errors = 0
STATISTICS FOR THE VDEVICE
tx_sent = 0
tx_dropped = 0
rx_received = 0
rx_dropped = 0
```

ii. ***vatm_ctl -c {type=atm} interface_id pvc=## {type=sar|dlc|sar+dlc|sar+qos+dlc} [local_esi_addr] [qos ubr/cbr:pcr=X]***

--to create a virtual device over ATM

This command is used to create the VATM. The options are defined below:

- *vatm_ctl -c* option is to create the VATM
- *{type=atm}* specifies the fact that the VATM would be created over an ATM card
- *interface_id* indicates the interface # of the ATM card
- *pvc=#* specifies the physical VCI with which the VATM would be hooked to the ATM card.
- *{type=sar|dlc|sar+dlc|sar+qos+dlc}* specifies the protocol stack for the VATM
- *[local_esi_addr]* specifies the ESI address
- *[qos ubr/cbr:pcr=X]* specifies the qos parameter for the physical VCI

iii. ***vatm_ctl -c {type=eth} interface_id {type=raw|mac|mac+rca} [local_esi_addr] [remote_esi_addr] {type=sar|dlc|sar+dlc|sar+qos+dlc}***

-- to create a virtual device over ETHERNET

- *vatm_ctl -c* option is to create the VATM
- *{type=eth}* specifies the fact that the VATM would be created over an Ethernet card
- *interface_id* indicates the interface # of the Ethernet card
- *{type=raw|mac|mac+rca}* specifies the mode of operation, raw and mac+rca is specific for the RDRN radios.
- *{type=sar|dlc|sar+dlc|sar+qos+dlc}* specifies the protocol stack for the VATM
- *[local_esi_addr]* specifies the local ESI address
- *[remote_esi_addr]* specifies the remote ESI address

iv. ***vatm_ctl -d interface***

-- to destroy a virtual device

v. ***vatm_ctl -i parameter value_of_parameter vatm_itf***

--to change the parameters of a layer
--parameter = DLC_SETFRAME_SIZE

A.2 Examples

A.2.1 To create a VATM (with SAR+DLC) on an ATM card

```
vatm_ctl -c atm 0 pvc=201 sar+dlc 0020ea000130 qos cbr:pcr=10Mbps
```

A.2.2 To create a VATM (with SAR only) on an ATM card

```
vatm_ctl -c atm 0 pvc=202 sar 0020ea000131 qos cbr:pcr=10Mbps
```

A.2.3 To create a VATM (with SAR+QoS+DLC) on an ATM card

```
vatm_ctl -c atm 0 pvc=203 sar+qos+dlc 0020ea000132 qos
cbr:pcr=10Mbps
```

A.2.4 To create a VATM (DLC only) on an ATM card

```
vatm_ctl -c atm 0 pvc=204 dlc 0020ea000133 qos cbr:pcr=10Mbps
```

A.2.5 To destroy a VATM

Let's us assume that the ITF of the VATM that we wish to destroy is 1.

```
vatm_ctl -d 1
```

A.2.6 To change the # of ATM cells that in a DLC frame

The following command would set the number of ATM cells in the DLC packet to 10 for the VATM whose ITF is 1

```
vatm_ctl -i DLC_SETFRAME_SIZE 10 1
```

A.2.7 To create the VATM and run it on the RDRN radios

Suppose we have the following configuration:

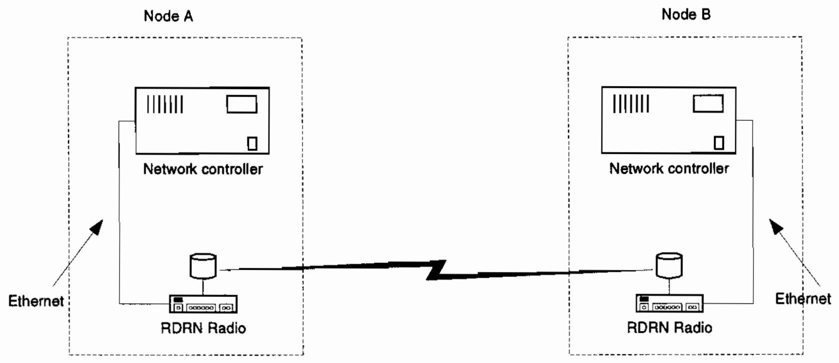


Figure 45: Running VATM on the RDRN radios

Let's say that ESI (End System Identifier) for the VATM on Node A is 0020ea000130 and on Node B is 0020eb000130

Hence to create the VATM with SAR+DLC protocol stack the following command needs to be executed:

Node A:

```
vatm_ctl -c eth 1 mac+rca 0020ea000130 00eb000130 sar+dlc
```

Node B:

```
vatm_ctl -c eth 1 mac+rca 0020eb000130 00ea000130 sar+dlc
```

Here, we have used "eth 1" since we have assumed that the network controller is connected to the radios on the second Ethernet card. If there were only 1 Ethernet card on the machine and that was the one connected to the radios, then it would have been "eth 0"

A.2.8 To set the credits on the VATM for the RDRN radios

This command is used to set the credits for the RDRN radios i.e. when the VATM is built on the Ethernet.

Suppose we have a VATM that is built on the eth1, then to set the credits to 2000 (say), the following command needs to be executed.

```
vatm_ctl -s eth1 credits 2000
```

Appendix B

B.1 Routing Algorithm

```
Program main
    do
        wait HI until
            ConnectMsg (k)
                Connect-To (k)
            LinkDownMsg (k)
                Lost-Link (k)
            Recv-Packet (p)
                Recv (p)
            Timeout-Event (e)
               Timeout-Handler (e)
         tiaw
     forever
end

Procedure Timeout-Handler (e)
    for each k in $B_x$
        if ($B^k_x$.timeout == e)
                    $B_x = B_x - k$
                    return
        fi
     rof
    for each k in $N_x$
        if ($N^k_x$.timeout == e)
                    Lost-Link (k)
                    return
        fi
    rof
   Send B_HELLO   //Broadcast
   Send-Updates ()
end

Procedure Forward-Packet (p)
    $R_{xj}$ = Find in Table (p.dst)
    min = infinity
    for each k in $R_{xj}$
        if(k($R^k_{xj}$) == p.src ) continue
        if(D($R^k_{xj}$) < min)
                    min = D($R^k_{xj}$)
                    n    = k($R^k_{xj}$)
        fi
     rof
     send p to n
end
```

```
Procedure Recv (p)
    If (p.port == ROUTER_PORT)
           Process-Update (p)
    Else
           Forward-Packet(p)
     Fi
End

Procedure Lost-Link (k)
      for each destination j
           Update ($R^k_{xj}$, infinity)
        Rof
end

Procedure Process-Update (p)
      switch(p.type )
             case B_HELLO :
                 if( k $\notin$ $B_x$) return
                 $B^k_x$.ctr ++
                 If($B^k_x$.ctr == 3)
                     Add-Nbr (k)
                     Send-Summary (k)
                  fi
                  return
              esac
              case N_HELLO :
                  if( k $\notin$ Nx )
                       Add-Nbr (k)
                       Send-Summary (k)
                   else
                       cancel ($N^k_x$.timeout )
                       $N^k_x$.timeout = schedule (3*HI)
                   fi
                   break
                esac
                case SMRY :
                    if( k $\notin$ Nx )
                         Add-Nbr (k)
                          Send-Summary (k)
                      else
                          if ( now - $N^k_x$.addtime > HI )
                               Send-Summary (k)
fi
                       fi
                   esac
      hctiws
      for each $U^k_j$ in p
           Verify-Update ($U^k_j$)
      rof
end
```

```
Procedure Verify-Update ($U^k_j$)
    if ($U^k_j$ D == INF )
        Update ($R^k_{xj}$ , $U^k_j$)
        return
    fi
    if ( ($U^k_j$ D == 0 && k $\in$ $N_x$) ||
        $U^k_j$ .D == $R^k_{xp}$.D )
        Update ($R^k_{xj}$ , $U^k_j$)
        return
    else
        $U^k_j$ .D = INF
        Update ($R^k_{xj}$ , $U^k_j$)
        return
    fi
end

Procedure Update ($R^k_{xj}$ , $U^k_j$ )
    for each n in $N_x$
        min_before[n]=Find-Min ($R_{xj}$ ,n)
    rof
    $R^k_{xj}$.D = $U^k_j$.D + 1
    $R^k_{xj}$.P = $U^k_j$.P
    for each k in $N_x$
        min_after = Find-Min ($R_{xj}$ ,n)
        if(min_after$\neq$min_before[n] ||
          min_before[n]==k)
            $N_{xj}$ = $N_{xj}$ + k
        fi
    rof
end

Procedure Send-Summary (k)
    New UPDATE ($U^k$)
    for each destination j
        $U^k_j$ =FindMin($R_{xj}$ ,k)
        Add $U^k_j$ to $U^k$
    Rof
    sort $U^k$
    send $U^k$ to k
end

Procedure Send-Updates ()
    for each k in $N_x$
        new UPDATE ($U^k$)
        for each destination j
            if(k $\in$ $N_{xj}$)
                $U^k_j$ = FindMin ($R_{xj}$ ,k)
                Add $U^k_j$ to $U^k$
                $N_{xj}$ = $N_{xj}$ – k
            Fi
        Rof
        sort $U^k$
        Send $U^k$ to k
    Rof
End
```

Table 7: Routing Algorithm[5]

B.2 Verify Routing Update & Update Table Algorithm

```
Procedure Verify-Update ($U^k_j$)
    // if reported distance to j is infinity
    if ( $D(U^k_j)$ == infinity )
        Update-Table ($R^k_{xj}$ , $U^k_j$)
        return
          fi

    // if we have an entry in the table for the
    // predecessor with the same reported
    // distance
    // or the reported distance is 0 and the
    //  destination is a neighbor
          if ( ( $D(U^k_j)$== 0 && $k \in N_x$) ||
          $D(U^k_j)$ == $D(R^k_{xp})$ )
        Update-Table ($R^k_{xj}$ , $U^k_j$)
        Return
    Else
        $U^k_j$ .D = infinity
        Update-Table ($R^k_{xj}$ , $U^k_j$)
        Return
          fi
end
```

```
Procedure Update-Table ($R^k_{xj}$ , $U^k_j$ )
    // find the shortest path information
    // that was reported to each neighbor
    // before updating the information
    for each n in $N_x$
            min_before[n]=Find-Min ($R_{xj}$ ,n)
    rof
    // update the entry
            $D(R^k_{xj}) = D(U^k_j) + 1$
            $P(R^k_{xj}) = P(U^k_j)$
    // find the shortest path information
    // to each neighbor after the update
    for each k in $N_x$
            min_after = Find-Min ($R_{xj}$ ,n)
    // if the shortest path information  changed
            if(min_after≠min_before[n] ||
               min_before[n]==k)
                  // add the neighbor to the list of
                  // neighbors to be updated
                  $N_{xj} = N_{xj} + k$
            fi
     rof
end
```

Table 8 Verify Routing Update & Update Table Algorithm[5]

B.3 Send Routing Update & Send Routing Summary Algorithm

```
Procedure Send-Updates ()
    // for each neighbor
    for each k in $N_x$
          new UPDATE ($U^k$)
          for each destination j
              // if neighbor k is affected by
              // changes in this entry
               if($k \in N_{xj}$)
                      $U^k_j$ = FindMin ($R_{xj}$ ,k)
                      Add $U^k_j$ to $U^k$
                      $N_{xj} = N_{xj} - k$
               fi
          rof
          sort  $U^k$
          Send $U^k$  to k
     Rof
End
```

```
Procedure Send-Summary (k)
    new UPDATE ($U^k$)
    for each destination j
          // Find shortest path not through k
            $U^k_j$ =FindMin($R_{xj}$ ,k)
            Add $U^k_j$ to $U^k$
    rof
    // sort the entries in the update by
    // distance
    sort  $U^k$
    send $U^k$ to k
end
```

Table 9 Send Routing Update & Send Routing Summary Algorithm[5]

Appendix C

C.1 An Example of Port-Map File

```
$ This is the KU port map file.
$
$ spud ports 1a1 - 1a4,
$            1c1 - 1c4
$            1d1 - 1d4
pm
  at:APM_Ver:1.0:string
  ar:atm_type:1
    ty:spud:ASX200bx:asx::
  ar:map_rdrn_atm:12
    mr:spud:1c2:testbed8:1a1:
    mr:spud:1d2:testbed9:1a1:
    mr:spud:1a4:testbed10:1a1:
    mr:spud:1a2:testbed11:1a1:
    mr:spud:1a3:testbed12:1a1:
    mr:spud:1c3:testbed13:1a1:
    mr:spud:1c4:testbed14:1a1:
    mr:spud:1a1:testbed15:1a1:
    mr:spud:1d1:testbed16:1a1:
    mr:spud:1d3:testbed17:1a1:
    mr:spud:1c1:testbed2:1a1:
    mr:spud:1d4:testbed5:1a1:
pe
```

C.2 An Example of the Scenario File

```
$ This is a 4-node scenario with 2 stationary MAPs, a stationary MEP
$ and a mobile MEP.
$ Text drawing of scenario. The dotted line is where the mobile node
$ move to.
$ O = MAP
$ X = MEP starting position
$ . = path

$ .                X
$     .
$         .   O
$             .
$                 .
$                     .
$                         .
$                             .
$                               X
$
$                      O

si
```

```
at:SDF_Ver:1.6:string
se

pi
at:platform_name:jeep1:string
at:comms_entity_type:MEP:string
ar:state:2
  st:0.0:-95.0:38.85:10.0:::::
  st:240.0:-95.5:39.0:10.0:::::
ep

pi
at:platform_name:jeep2:string
at:comms_entity_type:MEP:string
ar:state:1
  st:0.0:-95.25:38.99:10.0:::::
ep

pi
at:platform_name:truck1:string
at:comms_entity_type:MAP:string
ar:state:1
  st:0.0:-95.24:38.95:10.0:::::
ep

pi
at:platform_name:truck2:string
at:comms_entity_type:MAP:string
ar:state:1
  st:0.0:-95.1:38.86:10.0:::::
ep
```

Appendix D

D.1 Scenario File for Scenario 1 as shown in Section 4.1

```
$ This scenario uses 7 nodes. One mobile MAP, one mobile MEP, three
$ stationary
$ MAPS and two stationary MEPs.
$ Text drawing of scenario. The dotted line is where the mobile node
$ move to.
$ O = MAP starting position
$ X = MEP starting position
$ . = path

$       X..............................
$    O...................................
$
$         O           O           O
$       X                            X
$


si
at:SDF_Ver:1.6:string
se

$------------------------
$ Stationary MAP on left

pi
at:platform_name:MAP_west:string
at:comms_entity_type:MAP:string
ar:state:1
  st:0.0:-95.5:38.5:10.0:::::
ep

$-------------------------
$ Stationary MAP in middle
$ 7 km to the east of MAP_left

pi
at:platform_name:MAP_middle:string
at:comms_entity_type:MAP:string
ar:state:1
  st:0.0:-95.439798:38.5:10.0:::::
ep

$-------------------------
$ Stationary MAP on right
$ 8 km to the east of MAP_middle

pi
at:platform_name:MAP_east:string
at:comms_entity_type:MAP:string
```

```
ar:state:1
  st:0.0:-95.3681306:38.5:10.0:::::
ep

$-----------------------------
$ Mobile MAP
$ start 3.5 km west and 5 km north of MEP_left

pi
at:platform_name:MAP_mobile:string
at:comms_entity_type:MAP:string
ar:state:2
  st:0.0:-95.538518:38.550795:10.0:::::
  st:180.0:-95.287915:38.550795:10.0:::::
ep

$ Stationary MEP  on left

pi
at:platform_name:MEP_left:string
at:comms_entity_type:MEP:string
ar:state:1
  st:0.0:-95.533423:38.472250:10.0:::::
ep

$ Stationary MEP on right

pi
at:platform_name:MEP_right:string
at:comms_entity_type:MEP:string
ar:state:1
  st:0.0:-95.331157:38.478426:10.0:::::
ep


$ Mobile MEP

pi
at:platform_name:MEP_mobile:string
at:comms_entity_type:MEP:string
ar:state:2
  st:0.0:-95.508543:38.580909:10.0:::::
  st:180.0:-95.244704:38.586270:10.0:::::
ep
```

D.2 Scenario File for Scenario 2 as shown in Section 4.2[7]

```
$ This scenario uses 7 nodes. One stationary MEP, 1 mobile
$ MEP and 5 mobile MAPs.

$ Text drawing of scenario. The dotted line is where the mobile node
move to.
$ O = MAP starting position
$ X = MEP starting position
$ . = path

$

si
at:SDF_Ver:1.6:string
se

$------------------------
$ MAP at bottom left

pi
at:platform_name:MAP_1:string
at:comms_entity_type:MAP:string
ar:state:6
$ bottom left
  st:0.0:-95.25:39:10.0:::::
$ top left
  st:60.0:-95.2761:39.061:10.0:::::
$ top right
  st:120.0:-95.2075:39.1174:10.0:::::
$ bottom right
  st:180.0:-95.1359:39.0718:10.0:::::
  st:240.0:-95.1577:39.00017:10.0:::::
$ bottom -left
  st:300.0:-95.25:39:10.0:::::
ep

$-------------------------
$ MAP at top left

pi
at:platform_name:MAP_2:string
at:comms_entity_type:MAP:string
ar:state:6
$top left
  st:0.0:-95.2761:39.061:10.0:::::
$ top right
  st:60:-95.2075:39.1174:10.0:::::
$ bottom right
  st:120.0:-95.1359:39.0718:10.0:::::
  st:180.0:-95.1577:39.00017:10.0:::::
$ bottom left
  st:240.0:-95.25:39:10.0:::::
```

[7] The scenario here shows only the states of the node for the first 300 seconds only. After the 300 seconds, they repeat their states for the duration of the emulation period.

```
$ top left
  st:300.0:-95.2761:39.061:10.0:::::
ep

$-------------------------
$ MAP at top right

pi
at:platform_name:MAP_3:string
at:comms_entity_type:MAP:string
ar:state:6
  st:0.0:-95.2075:39.1174:10.0:::::
$ bottom right
  st:60.0:-95.1359:39.0718:10.0:::::
  st:120.0:-95.1577:39.00017:10.0:::::
$ bottom left
  st:180:-95.25:39:10.0:::::
$ top left
  st:240.0:-95.2761:39.061:10.0:::::
$top right
  st:300.0:-95.2075:39.1174:10.0:::::
ep

$-----------------------------
$ MAP at right bottom

pi
at:platform_name:MAP_4:string
at:comms_entity_type:MAP:string
ar:state:6
  st:0.0:-95.1359:39.0718:10.0:::::
  st:60.0:-95.1577:39.00017:10.0:::::
$ bottom left
  st:120:-95.25:39:10.0:::::
$ top left
  st:180.0:-95.2761:39.061:10.0:::::
$ top right
  st:240.0:-95.2075:39.1174:10.0:::::
$ bottom right
  st:300.0:-95.1359:39.0718:10.0:::::
ep

pi
at:platform_name:MAP_5:string
at:comms_entity_type:MAP:string
ar:state:6
  st:0.0:-95.1577:39.00017:10.0:::::
$ bottom left
  st:60:-95.25:39:10.0:::::
  st:120.0:-95.2761:39.061:10.0:::::
  st:180.0:-95.2075:39.1174:10.0:::::
$ top right
  st:240.0:-95.1359:39.0718:10.0:::::
$ bottom right
  st:300.0:-95.1577:39.00017:10.0:::::
$ top left
ep
```

```
$ MEP on left

pi
at:platform_name:MEP_left:string
at:comms_entity_type:MEP:string
ar:state:6
   st:0.0:-95.2646:39.0612:10.0:::::
   st:60.0:-95.2072:39.1084:10.0:::::
   st:120.0:-95.1475:39.0716:10.0:::::
   st:180.0:-95.1695:39.0105:10.0:::::
   st:240.0:-95.2387:39.0092:10.0:::::
   st:300.0:-95.2646:39.0612:10.0:::::
ep


$ MEP on right

pi
at:platform_name:MEP_right:string
at:comms_entity_type:MEP:string
ar:state:1
  st:0.0:-95.1241:39.063:10.0:::::
ep
```

References:

[1] *RDRN: A Rapidly Deployable Radio Network: Implementation & Experience* (ICUPC 98, Florence, Italy) Ricardo J Sanchez, Joseph B. Evans, Gary J. Minden, Victor S.Frost and K. Sam Shanmugam

[2] J.H. Condon et al. *Rednet: A Wireless ATM Local Area Network using Infrared Links*. In Proc. First International Conference on Mobile Computing and Networking (MOBICOM '95), November 1995.

[3] K.Y.Eng et al. *BAHAMA: A Broadband Ad-Hoc Wireless ATM Local-Area Network*. In Proc, International Conference on Communication (ICC '95), June 1995.

[4] L. French and D. Raychadhauri. *The WATMnet System: Rationale, Architecture, and Implementation*. International Proceeding IEEE Comp. Communication Workshop, September 1995.

[5] *Multi-Path Routing Protocol for Rapidly Deployable Radio Networks*, MS Thesis, University of Kansas, January 1999. Fadi Wahhab.

[6] *Linux ATM device driver interface Draft*, Werner Almesberger, Laboratoire de Reseaux de Communication (LRC) EPFL, Lausanne, Switzerland, February 1996.

[7] *RDRN Emulation Manager Documents*, Leon Searl, Swan Aircraft Inc., Available at http://www.ittc.ukans.edu/~searl

[8] *How to Configure Linux for ATM Networks*, Wayne Salamon, National Institute of Standards and Technology, June 1998.

[9] *Pthreads Programming*, Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell, O'Reilly & Associates, Inc.

[10] *ATM on Linux, User's Guide*, Werner Almesberger. Available at http://www.lrcwww.epfl.ch/linux-atm/.

Index