University of California
Santa Cruz

Routing in Packet-Switched Networks Using Path-Finding Algorithms

A dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy
in
Computer Engineering
by

Shree Murthy

September 1996

The dissertation of Shree Murthy is approved:

Prof. J. J. Garcia-Luna-Aceves

Prof. Patrick Mantey

Prof. Darrell Long

Dean of Graduate Studies and Research

Contents

List of Figures

List of Tables

Routing in Packet-Switched Networks Using Path-Finding Algorithms

Shree Murthy

Abstract

Route assignment is one of the operational problems of a communication network. The function of a routing algorithm is to guide packets through the communication network to their correct destinations. This dissertation is on the design and analysis of distributed, adaptive routing algorithms and protocols for packet switching networks. We introduce the general framework on which these algorithms are based. Using this general model, we propose several routing techniques to suit heterogeneous environments.

In this dissertation, we concentrate on distance-vector algorithms. One important drawback of previous distance-vector algorithms based on the distributed Bellman-Ford algorithm for shortest-path computation is that they suffer from counting-to-infinity problem and the bouncing effect. Recently, distributed shortest-path algorithms which utilize information about distance and second-to-last hop along the shortest-path to each destination have been proposed. This class of algorithms are called *path-finding algorithms*. Our proposals are based on path-finding algorithms.

We have proposed two new routing algorithms, PFA and LPA, for flat networks which are devoid of the drawbacks of the previous proposals. While PFA reduces the number of cases in which a temporary routing loop can occur, LPA is the first routing algorithm that is loop-free at every instant. To accommodate the increasing number of network users, aggregation of routing information is required. A hierarchical routing algorithm, HIPR, based on the maintenance and exchange of hierarchical routing trees, has been proposed for this purpose. To accommodate the low bandwidth requirements of mobile and wireless networks, a wireless routing protocol, WRP, has been proposed. This protocol minimizes protocol overhead.

To increase the responsiveness of a routing protocol and to guarantee the quality of service required by the user, routing and congestion control mechanisms have been integrated. This protocol ensures that packets arriving into a packet-switched network will be delivered unless a resource failure prevents it. Using this mechanism, we can ensure certain level of performance guarantees for network flows.

Acknowledgments

Several people supported me during my graduate study and I would like to thank them all. First of all, I would like to thank my advisor J.J. Garcia-Luna-Aceves for all his encouragement, patience and invaluable guidance. He was always there for me whenever I needed any sort of guidance. I feel very fortunate to have got an opportunity to work with him and I look forward to work with him in the coming years.

I would like to thank Anujan Varma for being in my proposal committee and for his valuable comments and suggestions. Many thanks to Patrick Mantey and Darrell Long for being in my committee and for their valuable suggestions.

Thanks are due to Bill Zaumen for answering all my questions on Drama very patiently and for making it possible for me to work with Drama for my simulations. Thanks are also due to Dr. Raphael Rom, for providing support and for bearing with my irregularities in the work schedule.

Thanks to the coco-group and especially to Jochen and Chane for making working in the lab so very enjoyable. Thanks to Atul, Hari, Vijay, Pratibha and Jyoti just for being there. Thanks to Jyoti, Hari and Ewerton for going through my dissertation and for their suggestions.

Thanks also to Nagesh, Ingrid for their encouragement and to little Kiran.

Most importantly, I would like to thank my parents for all their encouragement, love and support that has enabled me to live my dreams. Finally, I would like to thank God but for his grace, nothing would have been possible.

This work was funded in part by the office of Naval Research (ONR) under Grant No. N-00014-92-J-1807 and by the Defense Advanced Research Projects Agency (DARPA) under Grant No. DAAB07-95-C-D157.

Chapter 1
Introduction

One of the important components of a computer network is the communication subnetwork which includes the hardware and the software required for the transmission of data within the network. Traditionally, two different types of networks have supported the communication needs of end users: the telecom networks (telephone, cable and satellite networks) and data networks (Internet). The telephone network is designed to support high quality audio communications. These networks are engineered to provide low delay and fixed bandwidth service. Telecom networks are based on circuit-switching mode of operation in which, at the start of each session, the network determines the connection route and establishes end-to-end path from sender to receiver. In contrast, data networks are usually based on packet-switching, where there is no fixed physical path between a sender and a receiver. Instead, when a sender has a block of data to send, it is received in its entirety and then forwarded to the next hop along the path to the destination. Even though providing service guarantees is easier in circuit-switched mode of operation, because of the bursty nature of the traffic, packet-switching is favored in the present day Internet. Also, with the increasing demand for mobile and wireless communication, better techniques need to be developed for data transfer in a packet-switched environment.

In the context of the internal operation of a network, a connection is usually called *virtual circuit* in analogy with the physical circuits setup by the telephone system. The independent packets of the connectionless organization are called *datagrams*, in analogy with data networks. The idea behind a virtual circuit is to avoid having to make routing

Table 1.1: Comparison of Datagram and Virtual Circuit Networks

ISSUE	VIRTUAL CIRCUIT	DATAGRAM
Addressing	Each packet contains a short VC number	Each packet contains the source and the destination address
State Information	State information about each VC is maintained	Does not hold packet level state information
Routing	Route is chosen when VC is setup. All packets follow this route	Each packet is routed independently
Congestion control	Easy if enough buffers can be allocated in advance	Difficult
Resource failure	All VCs passing through the failed resource are terminated	Packets are lost only during resource failure
Suitability	Connection-oriented service	Connection-oriented and connectionless service

decisions for every packet sent. The route from a source to a destination is chosen as part of the connection setup mechanism.

In contrast, with a datagram network, no routes are setup in advance. Each packet is routed independently. Successive packets may follow different routes. While datagram networks have to do more work, they are more robust and adapt to congestion and failures easily. Table 1.1 summarizes some of the differences between datagrams and virtual circuits [Tan91].

Routing forms an integral part of the communications subnetwork. The *routing algorithm* is a part of the network layer which is responsible for deciding on which outbound queue an incoming packet should be transmitted. It guides packets through the communication network to their correct destinations. If the network uses datagrams internally, this decision must be made for every arriving data packet. However, if virtual circuits are used internally, routing decisions are made only when a new virtual circuit is being set up. The selection of the path towards a destination itself is made by a well-defined decision rule which is referred to as the *routing policy*.

Regardless of whether routes are chosen independently for each packet or only when new connections are established, certain properties are desirable in a routing algorithm. Some of them are:

- *Simplicity*: Simple algorithms are preferred for ease of implementation and higher efficiency in operational networks.
- *Robustness with respect to failures and changing conditions*: The algorithm must be able to adjust the routing decisions when traffic conditions change or when there is a resource failure. The algorithm monitors the network constantly and updates the routing information
- *Stability of the routing decisions*: The routing algorithms should adapt smoothly to changes in operating conditions. i.e., a small change in operating conditions should provide a comparatively small change in routing decisions.
- *Fairness of the resource allocation*: Data flows with the same characteristics should result in similar packet delay and throughput.
- *Optimality of the packet travel times*: The routing algorithm should maximize the network designer's objective function, while satisfying design constraints.
- *Loop freedom*: At any instant, the paths implied from the routing tables of all hosts taken together should not have loops. Each router in the path from a source to destination should be visited only once.
- *Convergence characteristics*: Time required to converge after a topology change should not be high. This is required to maintain up-to-date network state information.
- *Processing and memory efficiency*: The resources used at each router should be minimal. The computation time spent at a node affects the convergence time of the routing algorithm.

In this dissertation, our objective it to satisfy most of the above mentioned attributes of routing algorithms.

The two main functions performed by the routing algorithms are the selection of routes for various origin-destination pairs (*route computation*) and the delivery of messages of their correct destinations once the routes are selected (*packet forwarding*). We focus on the route computation function of the routing algorithm and design mechanisms to compute routes at each router.

Routing policies can be grouped into two major classes as *static* or *nonadaptive* and *adaptive* depending on whether the routes change in response to the current traffic pattern and topology. In a static routing policy, the path a packet takes from a source to a destination is predetermined. The routing tables are set up at a certain time before the data transmission begins and the routing tables are not changed thereafter. In an adaptive policy, packets are routed taking into account the current state of the network such that congested and damaged areas in the network are avoided. The routing tables are accordingly changed to dynamically adapt to changing network conditions. The information maintained at each routing node is updated taking into account the up-to-date network state information available at that time.

Adaptive routing algorithms require information about network traffic and topology to make good routing decisions. Depending on how and where this information is maintained, adaptive routing algorithms are further classified as *centralized* and *distributed* routing policies. In a centralized approach, the path information is computed at one central node, whereas in a distributed approach, routes are computed at each routing node using the network state information present at that node. Henceforth, we refer to *adaptive distributed routing* simply as *routing*.

Many practical routing algorithms are based on the notion of a *shortest path* between two nodes. Each communication link is assigned a positive number called its *length*. A link can have a different length in each direction. Each path (i.e., a sequence of links) between two nodes has a length equal to the sum of the lengths of its links. A shortest path routing algorithm routes each packet along a minimum length (shortest) path between the origin and destination nodes of the packet. The simplest possibility is for each link to have unit length (one hop), in which case a shortest path is simply a path with minimum number of links (hop count). More generally, the length may depend on transmission capacity and traffic load. The idea of shortest path algorithms is that the path should contain relatively few and uncongested links.

Distributed routing algorithms can be subdivided into *distance vector algorithms* (DVA) and *link-state algorithms* (LSA) depending on the method adopted to maintain routing information in router databases. In a distance vector algorithm, each node has knowledge of only the local links. The shortest paths are computed using a distributed version of Bellman-Ford algorithm [BG92] in which nodes exchange their shortest path lengths to other nodes with their neighbors periodically or on a event-driven basis. Using this information received from its neighbors, each node constructs a routing table containing the distance of the shortest path to every destination in the network. In other words, the process of route computation is carried out in a distributed way with each node performing part of the computation. Examples of distance-vector protocols include the old Arpanet algorithm [MW77], RIP [Hed88] and Cisco's IGRP [Hed].

In a link-state algorithm, each node has complete information of the network topology using which each node computes routes independently. When a node detects any change in the link distances, it sends out an update to all other nodes by broadcasting. Upon receiving an update, each node recomputes shortest paths to all other nodes using Dijkstra's shortest path algorithm [BG92] and constructs its new routing table. Some of the link state protocols are the new Arpanet routing protocol [MRR80], OSPF [Moy94] and ISO's IS-IS [Ora90].

1.1 Problem Formulation

In this dissertation, we concentrate on routing techniques for packet-switched networks using distance-vector algorithms. These algorithms are applicable to circuit-switched networks also. In the next few paragraphs we outline the motivation for our work. The main focus of this work is to identify the drawbacks of the existing routing techniques and propose routing algorithms to overcome these drawbacks. With this as the basis, the suitability of these algorithms to heterogeneous environments has been explored and the routing protocols have been proposed accordingly.

Some of the most popular routing protocols used in today's internets (e.g. RIP [Hed88]) are based on distributed Bellman-Ford (DBF) algorithm for shortest path computation [BG92]. However, DBF suffers from the *bouncing-effect* and *counting-to-infinity* problems. Recently, distributed shortest path algorithms that utilize information regarding the length and second to last hop of the shortest path to each destination have been proposed to overcome the counting-to-infinity problem of DBF. This class of algorithms is referred to as the *path-finding algorithms*. However, these algorithms do not eliminate the possibility of temporary routing loops. All the loop-free algorithms reported to date rely on mechanisms that require routers to either synchronize along multiple hops [GLA92a, JM82, MS79] or exchange path information that can include all the routers in the path from source to destination [GLA92b]. We propose two routing algorithms for a flat network topology that belong to the class of path-finding algorithms. The first of the two algorithms called Path-Finding algorithm (PFA), eliminates a number of cases in which a temporary routing loop can occur. The second algorithm, Loop-free Path-finding algorithm (LPA), guarantees loop-freedom at every instant.

Routing information maintained at each router must be updated frequently to dynamically adapt to the changes in the topology and congestion in the network. In an internetwork with a flat routing scheme, the size of the routing table grows linearly with the number of destinations in the network. With the increasing number of network users, aggregation of routing information becomes a necessity in any type of routing protocol. The goal of a hierarchical scheme is to reduce the size of the routing databases maintained at each router so that the exchange of routing or topology information among routers can be minimized. Prior proposals to hierarchical routing have assumed variants of DBF or topology broadcast algorithms. We propose a hierarchical routing algorithm based on the maintenance and exchange of hierarchical routing trees. We call this algorithm the *Hierarchical Information Path-based Routing* (HIPR). HIPR is based on a path-finding algorithm but does not require host routes for shortest path computation (unlike other path-finding algorithms) and is free of routing loops.

Today's internetwork technology is oriented towards computer communication in relatively stable operational environments. This cannot adequately support many of the emerging wireless applications. The challenge is to achieve reliable, high performance communications for mobile and wireless applications. Routing forms an integral part of this communications infrastructure. To adapt to the emerging applications, the routing protocols need to support wireless and mobile stations in addition to fixed stations. The routing protocols used in multihop packet radio network implemented in the past [Bea89, Bey90, LNT87] were based on shortest-path routing algorithms that have been typically based on DBF. DBF is susceptible to the counting-to-infinity problem and the bouncing-effect and will take a long time to converge. This is not desirable, more so in a wireless network since its bandwidth is very limited. Also, some of the techniques which have been proposed to overcome the basic problem of DBF in wired networks such as flooding, multihop internodal synchronization and the specification of complete path information would incur too much overhead with a dynamic topology and hence are not desirable. Therefore, a new routing protocol, devoid of all these drawbacks, is required to support the needs of emerging applications. We propose the wireless routing protocol (WRP) as a solution for routing in wireless networks. WRP overcomes the aforementioned drawbacks of DBF and exchanges minimal routing information among neighbors.

Routing and congestion control are two interrelated problems. Combining congestion control and routing techniques becomes especially important in order to guarantee quality-of-service requirements of the applications. A drawback of the existing Internet routing protocols is that their route computation and packet forwarding mechanisms are poorly integrated with congestion control mechanisms. More specifically, today's Internet routing is based on single-path routing algorithms. A routing protocol based on single-path routing is ill-suited to cope with congestion, because the only thing the protocol can do to react to congestion is to change the route used to reach a destination, and this could lead to unstable oscillatory behavior [Ber82]. In many networks there are several paths between pairs of nodes that are almost equally good. Better performance can be achieved by splitting the traffic over several paths to reduce the load on each of the links. This technique of using multiple routes between a single pair of nodes is called *multipath routing*. This is similar to inverse multiplexing in circuit-switched networks where the primary motivation is to provide high bandwidth with the limitation of low bandwidth links. Furthermore, in a datagram network, routers forward packets only on a best-effort basis and drop the packets when congestion occurs. The routers adapt to congestion only *after* network resources have already been wasted. We propose a new framework for dynamic multipath routing in packet-switched networks that attempts to prevent the over-utilization of network resources and hence avoid congestion. This protocol illustrates the provision of performance guarantees in a connectionless routing architecture. Using this approach, we propose a two-tier architecture in which the end users can request performance guarantees similar to a connection oriented architecture and, within the network, packet transmission is done hop-by-hop as in a connection-less network.

1.2 Dissertation Overview

This dissertation is organized as follows:

Chapter 2 gives an overview of the routing algorithms that are being used in today's internetwork. We highlight the problems in the existing algorithms and explain the working of the basic path-finding algorithm. We also introduce the network model and some of the terminologies used in the dissertation.

Chapter 3 presents two path-finding algorithms, *PFA* and *LPA*, which eliminate the looping problem of the existing distance-vector routing algorithms. We show through simulations that these two algorithms have better performance than the state of the art routing algorithms such as DUAL and an ideal link-state algorithm.

Chapter 4 proposes a novel methodology for routing in hierarchical networks. We formally verify the hierarchical routing algorithm and present some simulation results. The performance of this algorithm is compared with that of OSPF.

Chapter 5 describes a wireless routing protocol which is suitable in a packet-radio network. Simulation results of the basic routing algorithm are presented to evaluate the performance of the proposed protocol. Some implementation issues are also discussed.

Chapter 6 proposes a novel approach for integrating routing with congestion control. A worst-case delay bound for this dynamic solution is derived. A two-tier architecture is also proposed for mapping connection-oriented flows to connection-less flows and thereby guaranteeing certain level of QoS to end users of a packet-switched network.

Chapter 7 gives a summary of this work, together with some conclusions and directions for future research.

Chapter 2
Background

As explained in the previous chapter, routing algorithms are responsible for forwarding the data packets over routes to provide good or optimal performance. Consequently, a routing protocol is required to maintain the status of all the routes in the network. A router runs a specified routing algorithm to compute routes to all known destinations. A routing algorithm mainly consists of two parts – an initialization step and a recurring step that is repeated until the algorithm terminates. The recurring step involves updating the minimum distance of each router for all destinations until the algorithm converges to correct shortest path distances. The routing algorithms differ in the way by which the updating step is implemented. There are two types of adaptive routing algorithms – *link state* and *distance vector* algorithms.

Link-state Algorithms: In the link-state approach, each router maintains a complete view of the network topology and the cost associated with each link [MRR78, MRR80]. The topology information is updated regularly. A router broadcasts regularly the link state information of all its outgoing links to all other routers by flooding. A complete computation of the best routes is done at each node using the information present in its local topology database. When a router receives information about the change in the link cost, it updates its view of the network and applies a shortest path algorithm to choose its next hop to each destination.

Link state algorithms are basically free of long-term loops. Routers may not always have a consistent view of the network topology, because of the time updates take to reach all routers. This inconsistent view of the network can lead to the formation of loops, which are temporary and disappear in the time it takes for all routers to have the same topological information.

Link state algorithms have a disadvantage of not being scalable in terms of number of messages exchanged and the memory required to maintain the state of the entire network topology. Each time the topology changes, all network nodes have to recompute their routing tables, which creates a *peak* of activity.

Shortest Path First (SPF) [McQ74] is a link-state protocol in which each node computes and broadcasts the costs of its outgoing links periodically and applies Dijkstra's shortest path algorithm [BG92] to determine the next hop; other routing protocols that work on the same link-state approach are IS-IS [Ora90, Per91], and OSPF [Moy94].

Distance-Vector Algorithms: Distance-vector algorithms are often referred to as *Bellman-Ford* algorithms because they are based on the shortest-path computation algorithm by R.E. Bellman [Bel57]. Distance-vector algorithms have been used in several packet-switched networks such as Arpanet.

In a distance-vector algorithm, a router knows the length of the shortest-path (distance) from each of its neighbors to every destination in the network and uses this information to compute its own distance and the next router (successor) to each destination. Well-known examples of routing protocols which are based on distance-vector algorithms (DVA), are the routing information protocol (RIP) [Hed88], the HELLO protocol [Mil83a], the gateway-to-gateway protocol (GGP) [HS82], the exterior gateway protocol (EGP) [Mil83b] and the old Arpanet routing protocol [McQ74]. All these DVAs have used variants of the distributed Bellman-Ford algorithm (DBF) for shortest-path computation [BG92].

Distance-vector algorithms perform their route computation on a per-destination basis. If a link fails, only routes for those destinations which were routed over the failed link need to be recomputed. Moreover, the computation is localized to one part of the network only – the routers *upstream* of the failed link. Therefore, distance-vector algorithms are simpler.

The primary disadvantages of DBF are *routing-table loops* and *counting-to-infinity* problem [GLA]. A routing-table loop is a path specified in the routers' routing tables at a particular point in time, such that the path visits the same router more than once before reaching the intended destination. A router is said to be counting-to-infinity when it increments its distance to a destination until it reaches a predefined maximum distance value. Some solutions such as *split horizon* and *poisson reverse* have been proposed to overcome these basic problems [Hui95].

2.1 Network Model

A computer network G is modeled as an undirected graph represented as $G(V, E)$, where V is the set of nodes and E is the set of edges or links connecting nodes. Each node represents a router and is a computing unit involving a processor, local memory, and input and output queues with unlimited capacity. Extending the model to account for end node (link) destinations attached to routers is trivial. A functional bidirectional link connecting nodes i and j is denoted by (i, j) and is assigned a positive weight in each direction. A link is assumed to exist in both directions at the same time. All messages received (transmitted) by a node are put in the input (output) queue on a first-come-first-serve (FCFS) basis and are processed in that order. An underlying protocol assures that:

- Every node knows who its neighbors are; this implies that a node within a finite time detects the existence of a new neighbor or the loss of connectivity with a neighbor, or the change in the cost of an adjacent link.

- All packets transmitted over an operational link are received correctly and in the proper sequence within a finite time. (This assumption is made for convenience. Reliable message transmission can be easily incorporated into the routing protocol (e.g. [MGLA95, Moy94])

- All update messages, changes in the link-cost, link failures and link recoveries are processed one at a time in the order in which they occur.

Each node is given a unique identifier. Any link cost can vary over time but is always positive. The distance between the two nodes in the network is measured as the sum of the link costs of the shortest path between nodes.

When a link fails, the corresponding distance entry in a node's distance and routing tables are marked as infinity. A node failure is modeled as all links incident on that node failing at the same time. A change in the operational status of a link or a node is assumed to be notified to its neighboring nodes within a finite time. These services are assumed to be reliable and are provided by the lower level protocols.

Routing updates which are sent by a router to all its neighboring nodes can be of two types – *periodic* (time driven) and *triggered* (event driven). Periodic routing updates are sent periodically when the periodic update timer expires. The value of this timer depends on the network propagation delay and latency. Triggered updates increases the responsiveness of the protocol by requesting routers to send updates as soon as certain event occurs. Typical events are the changing of a local metric value, or the reception of a routing table update from a neighbor. This procedure speeds up the convergence time of the routing algorithm. The algorithms we propose use event-driven updating mechanism.

2.2 Notations and Definitions

A *path* from node i to node j is a sequence of nodes where (i, n_1), (n_x, n_{x+1}), (n_r, j) are links in the path. A *simple path* from i to j is a sequence of nodes in which no node is visited more than once. A *implicit path* from i to j is a path that is derived from predecessor node information. The paths between any pair of nodes and their corresponding distances change over time in a dynamic network. At any point in time, node i is connected to node j if a path exists from i to j at that time. The network is said to be connected if every pair of operational nodes are connected at a given time.

Throughout the paper the following notation is used:

$\emptyset$	: An empty set.
∞	: An arbitrarily large number.
$null$	: A nonexistent node.
A_i	: Set of areas in a hierarchical network
$C_j(t)$	: Loop formed for destination j at time t
D^i_j	: Distance entry at node i to destination j in the routing table
D^i_{jk}	: Distance entry at node i to destination j through neighbor k in the distance table
$FD^i_j(t)$	: Distance value used by node i to evaluate feasibility at time t
$H(I,d)$	: Maximum number of links in the loop-free path from node i having a length not exceeding d in the final topology
$LIST_k$	: List of entries received by node i in message M_k.
$LIST_i(n)$	: List of entries in a message M_i sent by node i to node n.
M_i	: Message sent by node i.
N_i	: Set of neighbors of i
$P_{xj}(t)$	: Path from node x to node j implied by successor entries at time t
$RD^i_j(t)$	: Distance from node i to node j at time t
$RH^i_j(t)$	: Predecessor of node j along the path from i to j at time t
$S_j(t)$	: Successor graph of G at i for destination j at time t
$T(i)$	: Time by which all messages that are in transit at time $T(i-1)$ have reached the destination
b, k	: Neighbor nodes
d_{ik}	: Link cost from i to neighbor k
j	: Destination node identifier $j \in N$
p^i_j	: Predecessor entry from i to j in the routing table
p^i_{jk}	: Predecessor entry from i to j through k in the distance table
r^i_{jk}	: Reply status flag for a query sent by node i for j through k

s_j^i : Successor from node i to j

$tag_j^i(t)$: Tag at node i for destination j at time t

$u_j^i(t)$: Update flag

The time at which the value of a variable applies is specified only when it is necessary. The successor to destination j for any node is simply referred to as the successor of that node, and the same reference applies to other information maintained by a node. Similarly, updates, queries and responses refer to destination j, unless stated otherwise.

In the algorithm's description, the time at which the value of a variable X of the algorithm applies is specified only when it is necessary; the value of X at time t is denoted by $X(t)$.

2.3 Evolution of Distance-Vector Algorithms

One of the earliest implementations of DVA was the routing protocol implemented in the Arpanet in the early 1970s. In this protocol, every router in the network maintains a distance and a routing table. The shortest path information for all destinations is maintained in the routing table. A router examines its routing table entries to determine the shortest path to a particular destination before sending a packet to that destination.

Many approaches have been proposed in the past to solve, at least in part, the looping problems in DVAs. A widely known proposal is the *split-horizon* technique, which avoids ping-pong looping, whereby two nodes choose each other as the successor to a destination [Ceg75, Sch86]. Another well known technique which has been proposed is the use of hold downs. Both of these approaches do not completely solve the counting-to-infinity problem [GLA]. Some other solutions have also been proposed to overcome this problem [GLA].

Figure 2.1 illustrates the looping and bouncing effect scenarios. Consider a three-node network with $n1$ being the destination node. Assume initially all nodes maintain correct routing table entries. Nodes $n2$ and $n3$ choose nodes $n1$ and $n2$ as their successors respec-

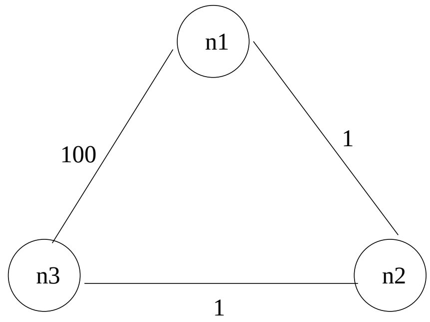

Figure 2.1: Counting-to-Infinity Problem

tively. Now, if link $(n1, n2)$ fails, based on the distance table entries, node $n2$ will choose $n3$ as its successor to destination $n1$. This information is sent to $n3$ (i.e., $n2$ announces a distance of 3 to reach $n1$), which leads to the formation of a routing loop between nodes $n2$ and $n3$. Furthermore, since the distances of $n2$ and $n3$ are much less than 100, which is the cost of the link $(n1, n3)$, nodes $n2$ and $n3$ will keep increasing their distances till a distance value > 100 is reached. After this, the distance converges and the correct path is chosen. Thus, we can see that, using DBF, nodes have to go through a long period of message exchanges among nodes belonging to loops before the algorithm converges. This is referred to as the *counting-to-infinity* problem.

Some of the most popular routing protocols used in today's Internet (e.g., RIP [Hed88]) are based on the distributed Bellman-Ford algorithm (DBF) for shortest-path computation [BG92]. The counting-to-infinity problem is overcome in one of the three ways in existing Internet routing protocols. OSPF [Moy94] relies on broadcasting complete topology information among routers, and organizes the Internet hierarchically to cope with the overhead incurred with topology broadcast. BGP [RL94] exchanges distance vectors that specify complete paths to destinations. EIGRP [Far93] uses a loop-free routing algorithm called DUAL [GLA92a], which is based on internodal coordination that can span multiple hops; DUAL also eliminates temporary routing loops.

Recently, distributed shortest-path algorithms [CRKGLA89, Hag83, Hum91, RF91, Mur94] that utilize information regarding the length and second-to-last hop (predecessor

Dest	Routing Table of n1
n1	0, *, *
n2	3, n6, n5
n3	4, n6, n2
n4	6, n6, n7
n5	2, n6, n6
n6	1, n6, n1
n7	5, n6 n3

(a)

Dest	Routing Table of n1
n1	0, *, *
n2	3, n6, n5
n3	4, n6, n2
n4	6, n6, n7
n5	2, n6, n6
n6	1, n6, n1
n7	5, n6 n3

(b)

Dest	Routing Table of n1
n1	0, *, *
n2	3, n6, n5
n3	4, n6, n2
n4	6, n6, n7
n5	2, n6, n6
n6	1, n6, n1
n7	5, n6 n3

(c)

Dest	Routing Table of n1
n1	0, *, *
n2	3, n6, n5
n3	4, n6, n2
n4	6, n6, n7
n5	2, n6, n6
n6	1, n6, n1
n7	5, n6 n3

(d)

Figure 2.2: Path Traversal using Predecessor Information

or node next to the last hop) of the shortest path to each destination have been proposed (path-finding algorithms) to eliminate the counting-to-infinity problem of DBF.

2.4 Path-Finding Algorithms

Path-finding algorithms eliminate the counting-to-infinity problem of DBF using predecessor information. Predecessor information can be used to infer an implicit path to a destination. Using this path information, routing loops can be detected. Each distance entry in the distance and routing tables is associated with the predecessor node information. The design of the path-finding algorithm is such that at all times, the distance and routing table entries satisfy the following property:

> *The path implicit in a distance entry from router i to destination j through a neighbor k, D^k_{ij}, with associated predecessor $h^k_{ij} = h$, is the path implicit to node h, D^k_{ih}, augmented by link (h, j).*

If each column in the distance and routing tables of a router satisfies this property at all times, then it can be used to maintain only simple paths to destinations.

Figure 2.2 illustrates the path traversal using predecessor information. Let $n1$–$n7$ be the nodes in a network. The figure shows the routing table entries at node $n1$. A routing table is a vector with each entry specifying the destination j, current shortest distance D^i_j, successor s^i_j and the predecessor p^i_j. Infinite distance is represented as ∞ and null path by *.

Suppose node $n1$ want to determine if its neighbor $n7$ is in the shortest path to destination $n2$. Node $n1$ starts the trace from the entry for destination $n2$ (Figure 2.2(a)) and finds that the predecessor to $n2$ is node $n5$. Subsequently, $n1$ walks through the predecessors of its path to $n5$ and $n6$ until it reaches node $n1$ itself (Figure 2.2(d)). From this, node $n1$ determines $n7$ is not in the path from $n1$ to $n2$ (not encountered during the trace). The sequence of predecessors encountered during such a trace represents a path from $n1$ to $n2$. This is referred to as the *implicit path* or the path extracted from the predecessor node information [CRKGLA89].

Although path-finding algorithms provide a marked improvement over DBF, the existing path-finding algorithms [CRKGLA89, GLA86, Hag83, Hum91, RF91] do not eliminate the possibility of temporary loops. The algorithms we have proposed are similar to previous path-finding algorithms with respect to maintaining predecessor information in distance and routing tables. Because each router reports to its neighbors the predecessor to each destination, any router can traverse the path specified by the predecessors from any destination back to a neighbor router to determine if using that neighbor as its successor would create a path that contains a loop (i.e., involves the router itself). Furthermore, a router detects a temporary loop within a finite time that depends on the speed with which correct predecessor information reaches the router, and not on the distance values of the paths offered by its neighbors; therefore, temporary loops are detected much faster than in DBF and its variations.

The next chapter describes the two algorithms, path-finding algorithm (PFA) and loop-free path-finding algorithm (LPA) and present some of their performance results. This forms the basis of our discussion in this dissertation.

Chapter 3
New Path-Finding Algorithms

This chapter describes two routing algorithms which we propose for a flat network. The two algorithms, path-finding algorithm (PFA) and loop-free path-finding algorithm (LPA), belong to the class of path-finding algorithms which forms the basis of our discussion in the rest of the dissertation. The working of the basic path-finding algorithm has been explained in the previous chapter.

3.1 Path-Finding Algorithm

PFA uses predecessor information to extract implicit paths from its distance and routing tables without excessive overhead. It substantially reduces the number of cases in which routing loops can occur. Each node maintains the shortest-path spanning tree reported by its neighbors, and uses this information and the information regarding the cost of the adjacent links to generate its own shortest-path spanning trees. The fact that PFA reduces temporary looping accounts for its superior performance over DUAL and the ideal link state algorithm (ILS). In addition to this, PFA also has an efficient updating mechanism. Each time an update is received by a router, the distance table and routing table entries are updated to reflect the change in the network state and thus maintain correct path information to all destinations.

3.1.1 PFA Description

Each node maintains a *distance table*, a *routing table* and a *link-cost table*. The *distance table* at node i is a matrix containing the distance (D^i_{jk}) and predecessor (p^i_{jk}) entries (path information) for all destinations (j) through all its neighbors (k). The *routing table* is a column vector of minimum distance to each destination (d^i_j) and its corresponding predecessor (p^i_j) and successor (s^i_j) information. The *link-cost* table lists the cost of each link adjacent to the node (l_{ik}); the cost of a failed link is considered to be infinity. An update message contains the source and the destination node identifiers, and the distance and predecessor for one or more destinations.

When a node i receives an update message from its neighbor k regarding destination j, the distance and the predecessor entries in the distance table are updated (Step 1). A unique feature of PFA is that node i also determines if the path to destination j through any of its other neighbors $\{b \in N_i | b \neq k\}$ includes node k. If the path implied by the predecessor information reported by node b includes node k, then the distance entry of that path is also updated as $D^i_{jb} = D^i_{kb} + D^k_j$ and the predecessor is updated as $p^i_{jb} = p^k_j$. Thus, a node can determine whether or not an update received from k affects its other distance and routing table entries. Before updating the routing table, node i checks for all simple paths to j reported by its neighbors, and the shortest of these simple paths becomes the path from i to j. This implies that at each stage node i checks for the simple paths and avoids loops. Link or node failures, recoveries and link-cost changes are handled similarly.

In contrast to PFA, which makes a node i check the consistency of predecessor information reported by *all* its neighbors each time it processes an event involving a neighbor k, all previous path-finding algorithms [CRKGLA89, Hum91, RF91] check the consistency of the predecessor only for the neighbor associated with the input event. This unique feature of PFA accounts for its fast convergence after a single resource failure or recovery as it eliminates more temporary looping situations than previous path-finding algorithms.

The following example illustrates the working of the algorithm. Consider a four node network shown in Figure 3.1(a). Let PFA be used at each node in this network. All links

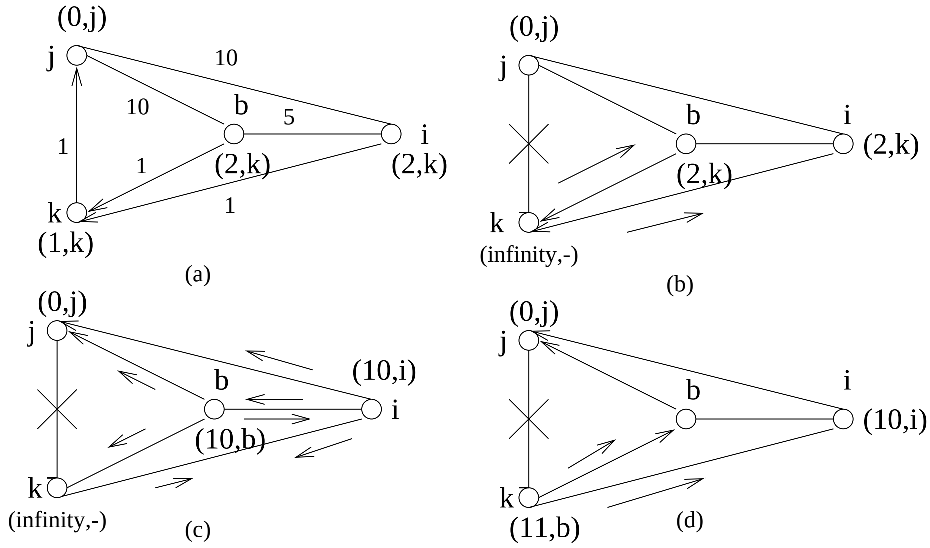

Figure 3.1: Example of PFA's Operation

and nodes are assumed to have the same propagation delay. Link-costs are as indicated in the figure and are assumed to be the same in both the directions. Node i is the source, j is the destination and nodes k and b are the neighbors of node i. The directed lines next to links indicate the direction of update messages and the label in parentheses gives the distance and the predecessor to destination j. This figure focuses on update messages to destination j only.

When link (j, k) fails, nodes j and k send update messages to their neighboring nodes as shown in Figure 3.1(b). In this example, node k is forced to report an infinite distance to j as nodes b and i have reported node k as part of their path to destination j. Node b processes node k's update and selects link (b, j) to destination j. This is because of the efficient updating mechanism of PFA that forces node b to purge any path to node j involving node k. Also, when i gets node k's update message, i updates its distance table entry through neighbor k and checks for the possible paths to destination j through any other neighboring nodes. Thus, a node examines the available paths through its other neighboring nodes and updates the distance and the routing table entries accordingly. This results in the selection of the link (i, j) to the destination j (Figure 3.1(c)). When node i receives b's update reporting an infinite distance, node i does not need to update its routing

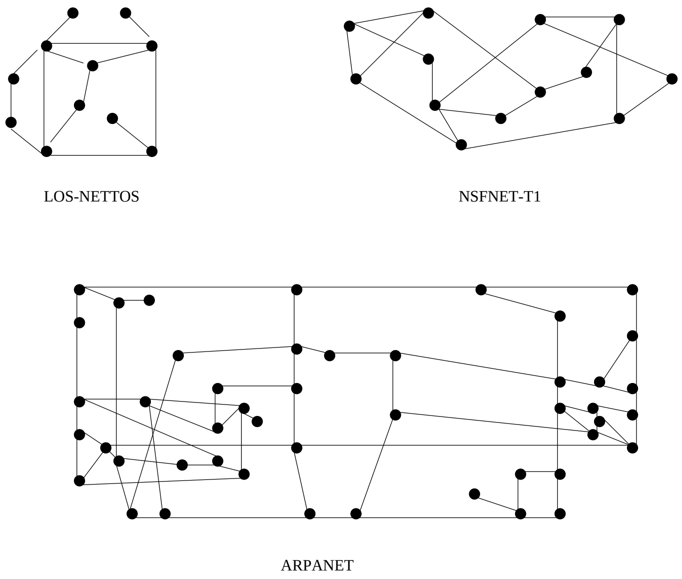

Figure 3.2: Simulated Topologies

table as it already has correct path information (Figure 3.1(d)). Similarly, updates sent by node k reporting a distance of 11 to destination j will not affect the path information of nodes i and b. This illustrates how the efficient updating mechanism of PFA helps in the reduction of the formation of temporary routing loops in the explicit paths.

The proofs of correctness, convergence and complexity of PFA are given elsewhere [Mur94]. The worst-case complexity of PFA has been found to be $O(h)$ for single recovery/failure, h being the height of the tree.

3.1.2 Simulation Environment

The performance of the proposed routing algorithms is evaluated by simulations. The simulation results of these algorithms have been compared with that of DUAL and an ideal link-state algorithm (ILS), which uses Dijkstra's shortest path algorithm for shortest path computation.

The *diffusing update algorithm* (DUAL) proposed by J.J. Garcia-Luna-Aceves aims at removing transient temporary loops. DUAL is based on the *diffusing* algorithm for partial route updates proposed by E.W. Dijkstra and C.S. Scholten in 1980 [DS80] and on the remark that one cannot create a loop by picking a shorter path to the destination [Jaf88].

DUAL uses an interneighbor coordination mechanism to achieve loop freedom. When there is no acceptable neighbor through which it can reach a given destination, the router will engage a diffusing computation. As long as this computation is not complete, the router freezes its routing tables, or at least the route to that destination. Since there are no loops before, freezing tables cannot create a loop. It however marks that destination as unreachable for the packets that are being sent towards the broken pipe, but this will last only for the duration of the diffusing computation.

Simulations have been developed using an actor-based, discrete-event simulation language called *Drama* [Zau91], together with a network simulation library. The library treats both links and routers as actors. Link failures and recoveries are simulated by sending a link status message to the routers at the end points of the appropriate links. Router failures can be simulated by making all the links connecting to that router to go down at the same time, and the link cost changes are treated as a link failing and recovering with a new cost.

All simulations are performed for unit propagation time. If a link fails, the packets in transit are dropped. A router receives a packet and responds to it by running the simulated routing algorithm and queueing the outgoing updates and processing the packets one at a time in the order of their arrival. The redundant packets are removed from the queue. The simulation ensures that all packets at a given simulation time are processed before the new updates are generated.

3.1.3 Instrumentation

A set of counters are used to instrument the simulations. These counters can be reset at various points. When the event queue empties that is, when the algorithm converges, the values of these counters are printed. During each simulation step, a router processes input events received during the previous step one at a time, and generates messages as needed for each input event it processes.

To obtain the statistical averages, the simulation makes each link (router) in the network fail, and counts the steps, messages and operations needed for each algorithm to converge.

It then makes the same link (router) recover and repeats the process. The average is then taken over all link (router) failures and recoveries. The routing algorithm was allowed to converge after each such change. In all cases, routers were assumed to perform computations in zero time and links were assumed to provide one time unit of delay. For the failure and recovery runs, the costs are set to unity. Both the mean and the standard deviation are computed for each counter; the four counters used are

- *Update Count:* The total number of updates (including queries and replies where applicable) and changes in link status processed by routers.
- *Message Count:* The total number of packets transmitted over the network. Each packet may contain multiple updates.
- *Duration:* The total elapsed time it takes for an algorithm to converge.
- *Operations:* The total number of operations performed by all nodes in the network. The operation count is incremented whenever an event occurs, but also counts the number of times the statement within a *for* or *(while) loop are executed.*

There is no sampling error for the results because all possible cases are covered in the simulations. Both the mean and the standard deviation of the distributions are given.

3.1.4 Simulation Results

The performance of PFA has been compared with DBF, DUAL and ILS. The simulations were run on several network topologies such as *Los-Nettos*, *Nsfnet* and *Arpanet* (Figure 3.2). We choose these topologies to compare the performance of routing algorithms for well-known cases, given that we cannot sample a large enough number of networks to make statistically justifiable statements about how an algorithm scales with network parameters. Here we present the simulation results of Arpanet topology only.

For the routing algorithms under consideration, there is only one shortest path between a source and a destination pair and we do not consider null paths from a router to itself. Data was collected for a large number of topology changes to determine statistical distributions.

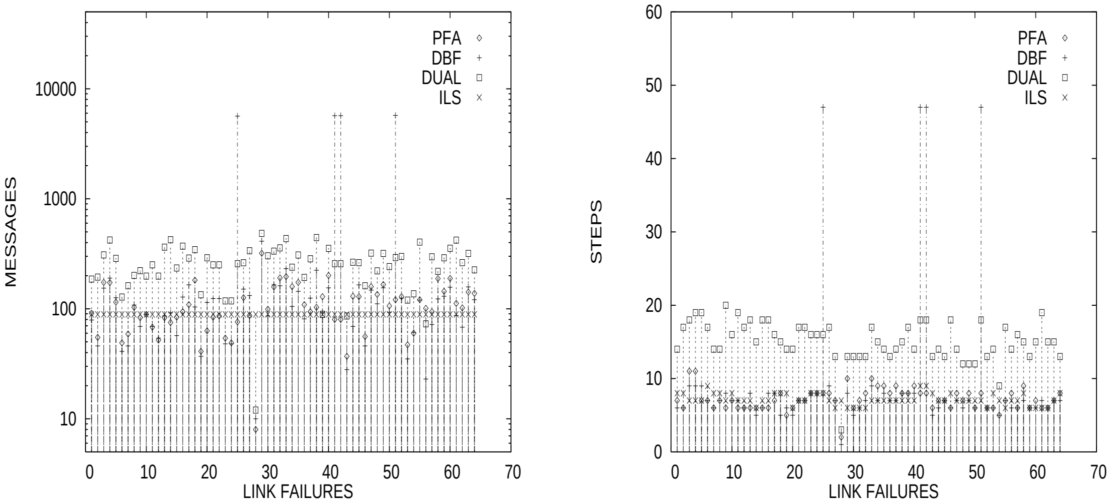

Figure 3.3: ARPANET Link Failure

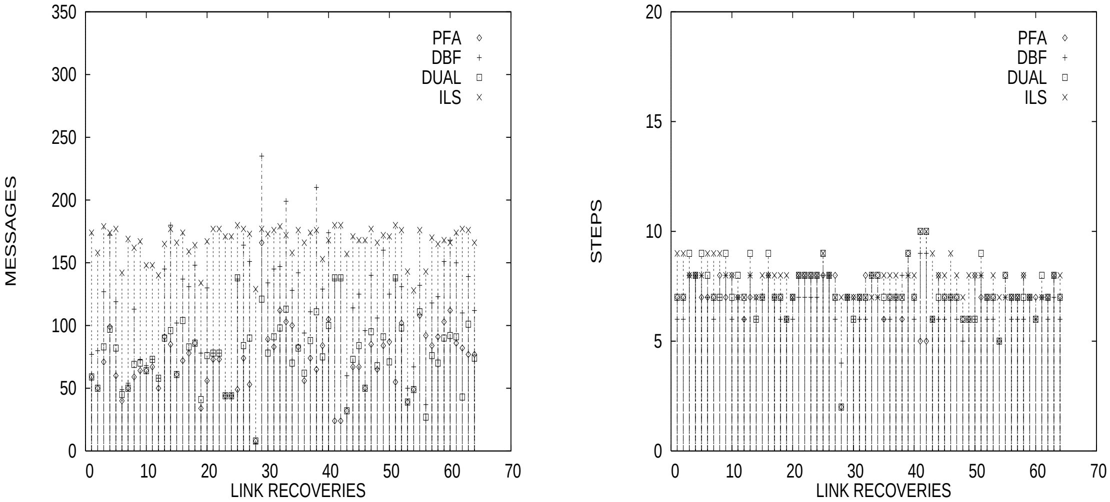

Figure 3.4: ARPANET Link Recovery

Total Response to a Single Resource Change

The graphs in Figures 3.3 and 3.4 depicts the number of messages exchanged and the number of steps required before each algorithm converges for every link failing and recovering in the Arpanet topology. Similar graphs for every router failing and recovering is given in Figures 3.5 and 3.6 respectively. All topology changes are performed one at a time and the algorithms were allowed to converge after each such change before the next resource change occurs. The ordinates of the graphs represents the identifiers of the links and the nodes while the data points show the number of messages exchanged after each resource change

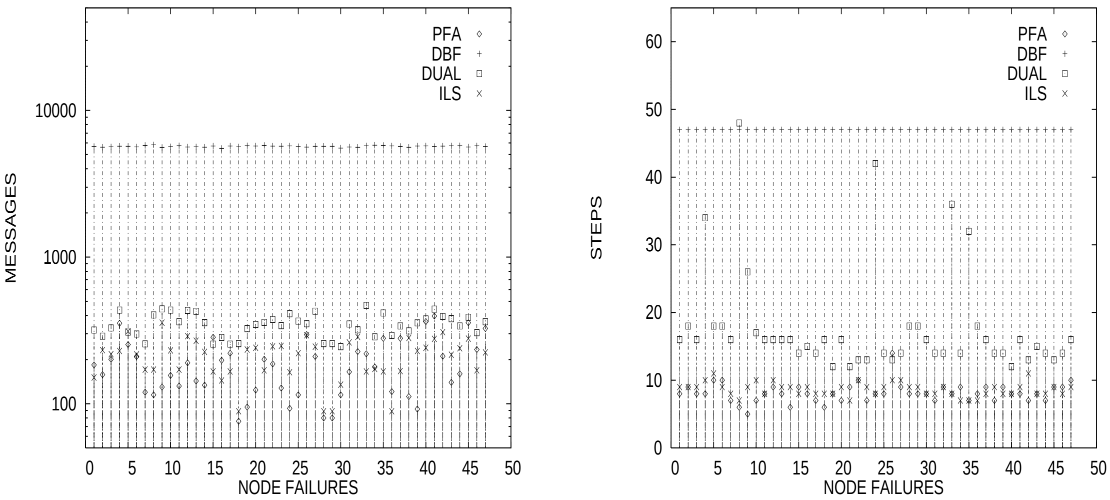

Figure 3.5: ARPANET Router Failure

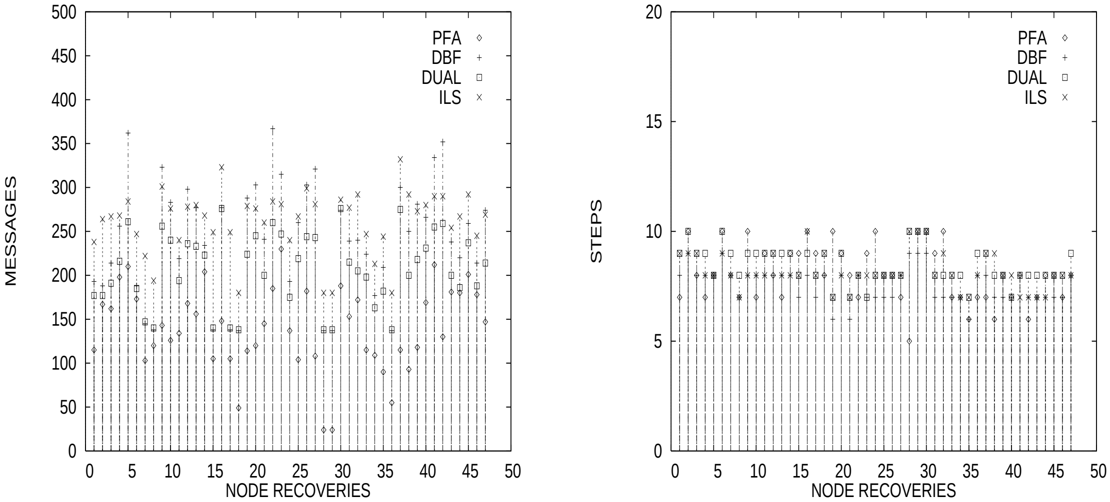

Figure 3.6: ARPANET Router Recovery

(graphs on the left hand side) and the number of steps needed for convergence (graphs on the right hand side) in each of these figures.

For a single resource failure, PFA outperforms both DBF and DUAL. This is because PFA does not suffer from the counting-to-infinity problem of DBF and does not use an internodal coordination mechanism that spans several hops to achieve loop freedom as in DUAL. The performance of PFA is comparable to ILS. The overall performance of the protocol is also comparable to ILS. The overall performance of PFA and DUAL after the recovery of a single router or a link is better than ILS. This is expected of any distance-vector algorithm. The convergence time of PFA and DUAL is also comparable to that of ILS. For a single resource recovery also PFA and DUAL are superior to ILS.

Dynamic Response to a Single Change

To study the dynamic behavior of the routing algorithms, we ran an exhaustive series of test cases for all router and link failures and recoveries and recorded the required statistics. A statistical characteristic was obtained by treating each router change as a separate case and by computing a distribution as a function of time. In this section, we present the results of the dynamic behavior of the above mentioned algorithms for *Arpanet* topology.

Instrumentation has been done to take care that a path from a router to itself is not considered. Some of the statistics also have been characterized by the probability as a function of time that some condition is true and by an average value given that a condition is true.

Figures 3.7, 3.9, 3.11 and Figures 3.8, 3.10, 3.12 show the transient response of the routing algorithm after a link failure and recovery respectively. Figures 3.13, 3.15, 3.17 and Figures 3.14, 3.16, 3.18 show similar graphs for router failure and recoveries respectively. The figures show the average packet length, the probability that the messages are in transit and the average number of messages that are exchanged after a resource change. All these parameters are plotted as a function of time.

The results indicate that for a resource failure, ILS performs better than DUAL and PFA in terms of the number of messages exchanged. However, the performance of PFA is comparable to ILS for a resource failure and performs much better than DUAL. For a resource recovery also, the performance of DUAL and PFA are comparable and performs better than ILS. The average packet length for PFA is smaller than DUAL. This is because of the *tagging scheme* used in PFA.

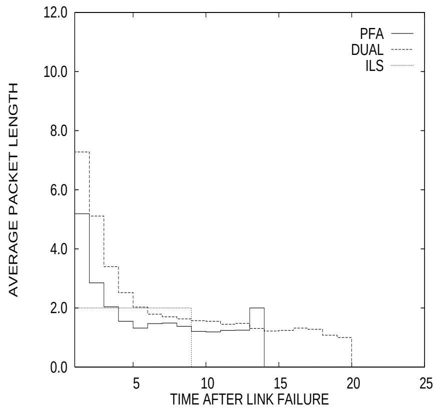

Figure 3.7: Avg pkt len for Link Failure

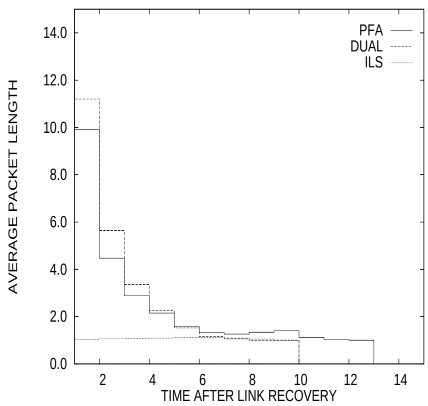

Figure 3.8: Avg pkt len for Link Recovery

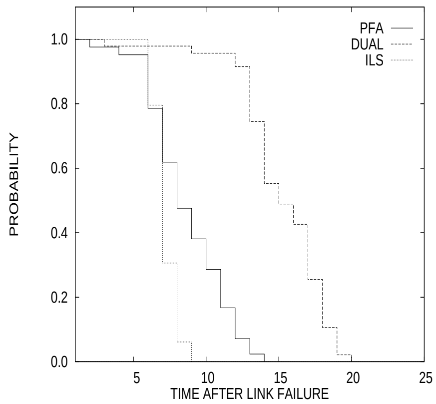

Figure 3.9: Prob of pkts for Link Failure

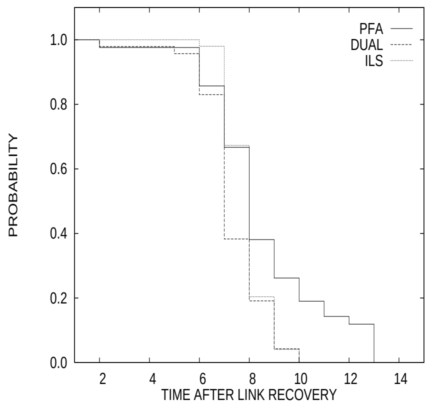

Figure 3.10: Prob of pkts for Link Recovery

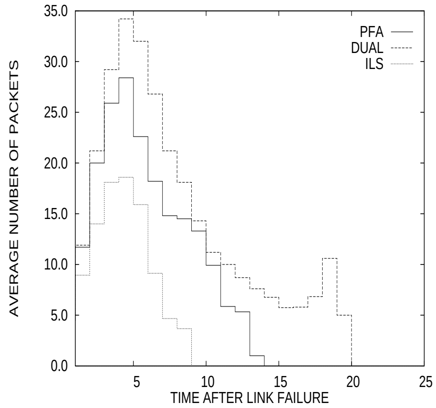

Figure 3.11: Avg num of pkts for Link Failure

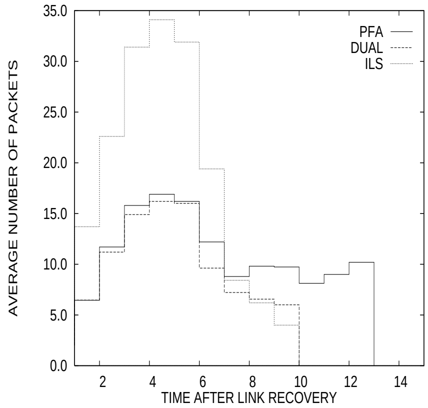

Figure 3.12: Avg num of pkts for Link Recovery

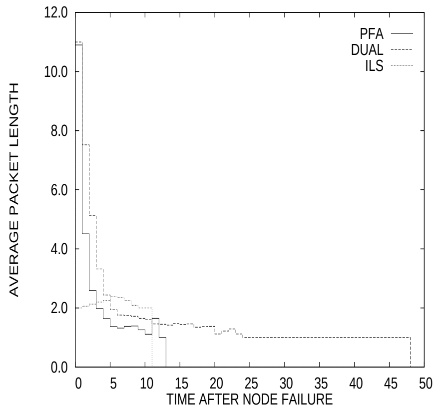

Figure 3.13: Avg pkt len for Router Failure

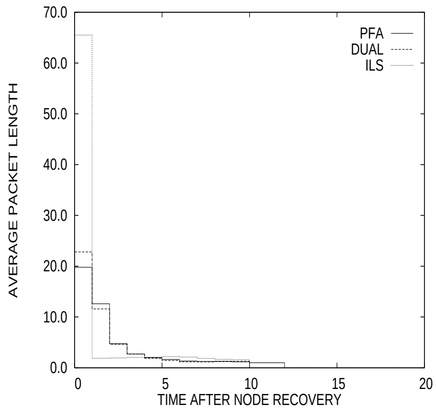

Figure 3.14: Avg pkt len for Router Recovery

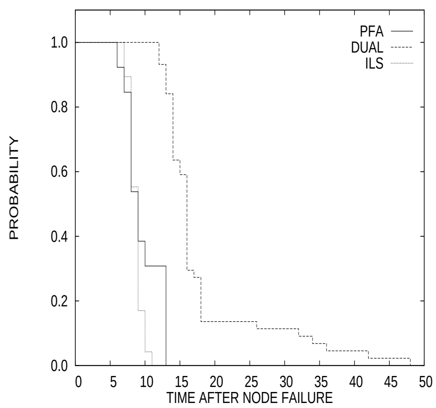

Figure 3.15: Prob of pkts for Router Failure

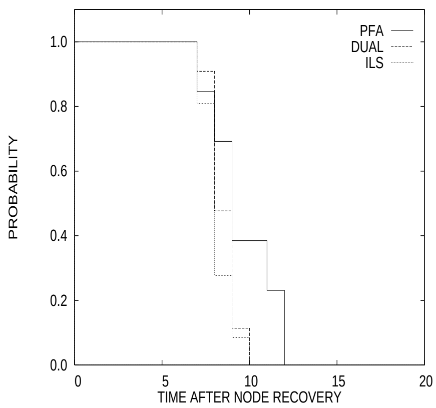

Figure 3.16: Prob of pkts for Router Recovery

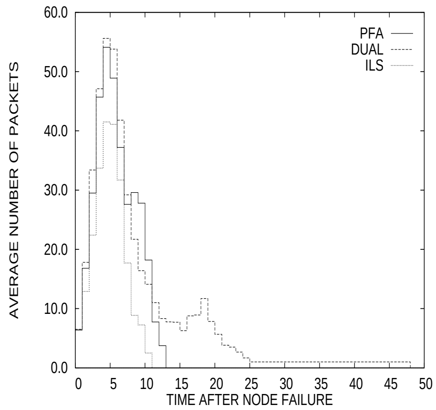

Figure 3.17: Avg num of pkts for Router Failure

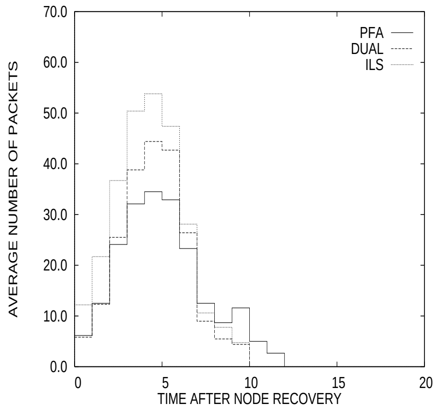

Figure 3.18: Avg num of pkts for Router Recovery

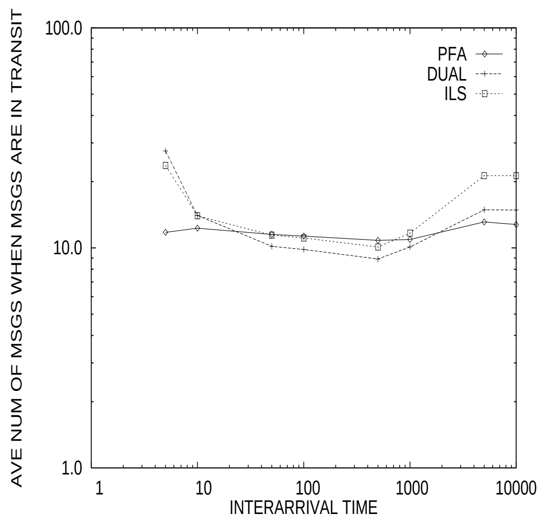

Figure 3.19: Average Number of Messages

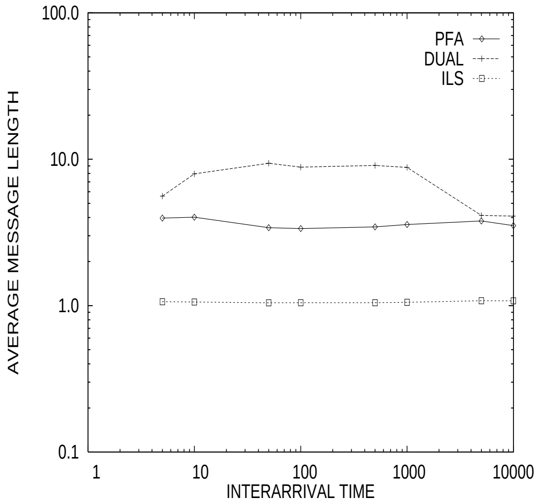

Figure 3.20: Average Message Length

Response to Multiple Link-Cost Changes

The steady-state behavior of the algorithms is more interesting with multiple link-cost changes than the transient response after each topology change. Figures 3.19 and 3.20 shows the average number of update messages when messages are in transit and the average length of messages as a function of the interarrival times between link-cost changes for PFA, DUAL and ILS respectively. This again is for Arpanet topology. From [ZGLA92], it has been observed that the behavior of DUAL and ILS for multiple link-cost changes is similar for different network topologies; our conjecture is that the same is true for PFA also.

For very long interarrival times, the number of messages during busy periods is independent of the interarrival time because the probability of two topology changes occurring simultaneously is small. In this case, the performance approaches that of single link-cost change. When the interarrival time approaches the network diameter, this situation changes and the number of messages during the busy period increases because of multiple topology changes occurring simultaneously.

The average number of messages exchanged when messages are in transit is slightly less for PFA compared to DUAL and ILS. This is because PFA does not use any interneighbor coordination mechanism to eliminate all temporary looping cases.

3.2 Loop-Free Path-Finding Algorithm

All the loop-free algorithms reported to date rely on mechanisms that require routers to either synchronize along multiple hops [GLA92a, JM82, MS79], or exchange path information that can include all the routers in the path from a source to destination [GLA92b]. We present the loop-free path-finding algorithm (LPA), which is the first routing algorithm that is loop-free at every instant and does not use either of these two techniques.

Updates take time to propagate and routers have to update their routing tables using information that can be out of date, which can lead to temporary routing loops. In LPA, when a router detects that it can create routing table loops if it changes its successor to a destination, it blocks such a potential loop. The router accomplishes this by reporting an infinite distance for the destination to all its neighbors and by waiting for those neighbors to acknowledge its message with their own distances and predecessor information, before the router changes its successor. Because of the overhead involved, a router should not send a query every time it has to change its successor to a destination; a router decides when to block a potential loop by comparing the distances reported by its neighbors against a *feasible distance*, which is defined to be the smallest value achieved by the router's own distance since the last query sent by the router. The router is forced to block a potential loop with a query only when no neighbor reports a distance smaller than the router's own feasible distance. This feature accounts for the low overhead incurred in LPA to accomplish loop-free paths at every instant. Furthermore, a router detects a temporary loop within a finite time that depends on the speed with which correct predecessor information reaches the router, and not on the distance values of the paths offered by its neighbors; therefore, temporary loops are detected much faster than in DBF and its variations.

3.2.1 LPA Description

Information Maintained and Exchanged

Each router maintains a *distance table*, a *routing table* and a *link-cost table*. The distance table at each router i is a matrix containing, for each destination j and for each neighbor k of router i, the distance and the predecessor reported by router k, denoted by D^i_{jk} and p^i_{jk}, respectively. The set of neighbors of router i is denoted by N_i.

The routing table at router i is a column vector containing, for each destination j, the minimum distance (denoted by D^i_j), the predecessor (denoted by p^i_j), the successor (denoted by s^i_j), and a marker (denoted by tag^i_j) used to update the routing table. For destination j, tag^i_j specifies whether the entry corresponds to a simple path ($tag^i_j = correct$), a loop ($tag^i_j = error$) or a destination that has not been marked ($tag^i_j = null$).

The link-cost table lists the cost of each link adjacent to the router. The cost of the link from i to k is denoted by d_{ik} and is considered to be infinity when the link fails.

An update message from router i consists of a vector of entries reporting incremental updates to its routing table; each entry specifies an update flag (denoted by u^i_j), a destination j, the reported distance to that destination (denoted by RD^i_j), and the reported predecessor in the path to the destination (denoted by rp^i_j). The update flag indicates whether the entry is an update ($u^i_j = 0$), a query ($u^i_j = 1$) or a reply to a query ($u^i_j = 2$). The distance in a query is always set to ∞.

Because every router reports to its neighbors the second-to-last hop in the shortest path to each destination, the complete path that a router assumes to any destination (called its implicit path to the destination) at a given time t' is known by the router's neighbors at a subsequent time $t'' > t'$. This is done by means of a path traversal routine on the predecessor entries reported by the router. In the specification of LPA, the successor to destination j for any router is simply referred to as the successor of the router, and the same reference applies to other information maintained by a router. Similarly, updates, queries and replies refer to destination j, unless stated otherwise. Figures 3.21 and 3.22

specify LPA in pseudo-code. The rest of this section provides an informal description of LPA.

The procedures used for initialization are *Init1* and *Init2*; Procedure *Message* is executed when a router processes an update message; procedures *linkUp*, *linkDown* and *linkChange* are executed when a router detects a new link, the failure of a link, or the change in the cost of a link. We refer to these procedures as event-handling procedures. For each entry in an update message, Procedure *Message* calls procedure *Update*, *Query*, or *Reply* to handle an update, a query, or a reply, respectively. An important characteristic of all event-handling procedures is that they mark $tag_j^i = null$ for each destination j affected by the input event.

Router i initializes itself in passive state with an infinite distance for all its known neighbors and with a zero distance to itself. After initialization, router i sends updates containing the distance to itself to all its neighbors.

Distance Table Updating

When router i receives an input event regarding neighbor k (an update message from neighbor k or a change in the cost or status of link (i,k)), it updates its link-cost table with the new value of link d_{ik} if needed, and then executes procedure DT. The intent of this procedure is for the router to erase the outdated path information in the distance table by making path information from all neighbors consistent with the latest update. To accomplish this, DT updates the distance and predecessor entries of neighbor k as $D_{jk}^i = D_j^k + d_{ik}$ and $p_{jk}^i = p_k^i$ for each destination j affected by the input event. In addition, DT determines whether the path to any destination j through any of the other neighbor of router i includes neighbor k. This is done by traversing the path specified by the predecessor entries reported by a neighbor from destination j towards node i. If the path implied by the predecessor reported by router b ($b \neq k$ and $b \in N_i$) to destination j includes router k, then i assumes that b has outdated path information and substitutes the subpath from k to j reported by b as part of its path to j with the path reported by k itself. This is done by updating $D_{jb}^i = D_{kb}^i + D_j^k$ and $p_{jb}^i = p_j^k$.

```
Procedure Init1
when router i initializes itself
do begin
    set a link-cost table with
    costs of adjacent links;
    N ← {i}; N_i ← {x | d_ix < ∞};
    for each (x ∈ N_i)
    do begin
        N ← N ∪ x; tag^i_x ← null;
        s^i_x ← null; p^i_x ← null;
        D^i_x ← ∞; FD^i_x ← ∞
    end
    s^i_i ← i; p^i_i ← i; tag^i_i ← correct;
    D^i_i ← 0; FD^i_i ← 0;
    for each j ∈ N call Init2(x, j);
    for each (n ∈ N_i) do
    add (0, i, 0, i) to LIST_i(n);
    call Send
end

Procedure Init2(x, j)
begin
    D^i_jx ← ∞; p^i_jx ← null;
    s^i_jx ← null; r^i_jx ← 0;
end

Procedure Send
begin
    for each (n ∈ N_i)
    do begin
        if (LIST_i(n) is not empty)
        then send message with
          LIST_i(n) to n
        empty LIST_i(n)
    end
end

Procedure Reply(j, k)
begin
    r^i_jk ← 0;
    if (r^i_jn = 0, ∀n ∈ N_i)
    then if ((∃x ∈ N_i | D^i_jx < ∞)
      or (D^i_j < ∞))
        then call PU(j)
    else call AU(j, k)
end

Procedure Message
when router i receives a message
  on link (i, k)
begin
    for each entry (u^k_j, j, RD^k_j, rp^k_j)
      such that j ≠ i
    do begin
        if (j ∉ N)
        then begin
            if (RD^k_j = ∞)
            then delete entry
            else begin
                N ← N ∪ {j}; FD^i_j = ∞;
                for each x ∈ N_i
                  call Init2(x, j)
                tag^i_j ← null;
                call DT(j, k)
            end
        end
        else
            tag^i_j ← null; call DT(j, k)
    end
    for each entry (u^k_j, j, RD^k_j, rp^k_j) left
    such that j ≠ i
    do case of value of u^i_j
        0:  [Entry is an update]
            call Update(j, k)
        1:  [Entry is a query]
            call Query(j, k)
        2:  [Entry is a reply]
            call Reply(j, k)
    end
    call Send
end

Procedure Update(j, k)
begin
    if (r^i_jx = 0, ∀x ∈ N_i)
    then begin
        if ((s^i_j = k) or (D^i_jk < D^i_j))
        then call PU(j)
    end
    else call AU(j, k)
end

Procedure PU(j)
begin
    DT_min ← Min{D^i_jx ∀ x ∈ N_i};
    FCSET ← {n | n ∈ N_i, D^i_jn = DT_min,
      D^n_j < FD^i_j};
    if (FCSET ≠ ∅) then begin
      call TRT(j, DT_min);
      FD^i_j ← Min{D^i_j, FD^i_j}
    end
    else begin
      FD^i_j = ∞; r^i_jx = 1,   ∀x ∈ N_i;
      D^i_j = D^i_{j s^i_j};
      p^i_j = p^i_{j s^i_j};
      if (D^i_j = ∞) then s^i_j ← null;
      ∀ x ∈ N_i do begin
        if (query and x = k)
        then r^i_jk ← 0;
        else add (1, j, ∞, null)
          to LIST_i(x)
      end
    end
end

Procedure Query(j, k)
begin
    if (r^i_jx = 0∀x ∈ N_i)
    then begin
        if (D^i_j = ∞ and D^i_jk = ∞)
        then add (2, j, D^i_j, p^i_j)
          to LIST_i(k)
        else begin
            call PU(j);
            add (2, j, D^i_j, p^i_j)
              to LIST_i(k);
        end
    else call AU(j, k)
end
```

Figure 3.21: LPA Specification

The example in Figure 3.23 illustrates how procedure DT helps to expedite LPA's convergence. In the example, router x has reported to i its predecessors to y and j, and i infers that x's path to j is xyj. Router c has reported to i its predecessors to d, a, b and j, and i infers that c's path to j is $cdabj$. Router a has reported to i its predecessors to b and j, and i infers that a's path to j is abj. With these conditions, assume that a sends i an update stating that $D^a_j = \infty$ and $p^a_j = null$. Router i uses procedure DT to ensure that the path information from the other neighbors reflects the most recent update obtained from any other neighbors for any destination. Since c's path to j includes a, i substitutes the out-of-date sub-path abj in c's path information with the information supplied by a, which makes the path from c to j non-existent. The path from x to j does not include a and i

```
Procedure Link_Up (i, k, d_ik)
when link (i, k) comes up do begin
    d_ik ← cost of new link;
    if (k ∉ N) then begin
        N ← N ∪ {k}; tag^i_k ← null;
        D^i_k ← ∞; FD^i_k ← ∞;
        p^i_k ← null; s^i_k ← null;
        for each x ∈ N_i do call Init2(x, k)
    end
    N_i ← N_i ∪ {k};
    for each j ∈ N do call Init2(k, j);
    for each j ∈ N − k | D^i_j < ∞ do
        add (0, j, D^i_j, p^i_j) to LIST_i(k);
    call Send
end

Procedure Link_Down(i, k)
when link (i, k) fails do begin
    d_ik ← ∞;
    for each j ∈ N do begin
        call DT(j, k);
        if (k = s^i_j) then tag^i_j ← null
    end
    delete column for k in distance table;
    N_i ← N_i − {k};
    delete r^i_jk;
    for each j ∈ (N − i) | k = s^i_j
        call Update(j, k)
    call Send
end

Procedure AU(j, k)
begin
    if (k = s^i_j) then begin
        D^i_j ← D^i_jk; p^i_j ← p^i_jk;
    end
end

Procedure DT(j, k)
begin
    D^i_jk ← RD^k_j + d_ik; p^i_jk ← rp^k_j;
    for each neighbor b do begin
        h ← j;
        while (h ≠ i or k or b) do h ← p^b_h;
        if (h = k) then begin
            D^i_jb ← D^i_kb + RD^k_j; p^i_jb ← rp^k_j;
        end
        if (h = i) then begin
            D^i_jb ← ∞; p^i_jb ← null;
        end
    end
end

Procedure Link_Change (i, k, d_ik)
when d_ik changes value do begin
    old ← d_ik;
    d_ik ← new link cost;
    for each j ∈ N do begin
        call DT(j, k);
        for each j ∈ N
        do if (D^i_j > D^i_jk or k = s^i_j)
        then tag^i_j ← null;
    end
    for each j ∈ N do begin
        if (d_ik < old)
        then for each j ∈ N − i | D^i_j > D^i_jk
            do call Update(j, k)
        else for each j ∈ N − i | k = s^i_j
            do call Update(j, k)
    end
    call Send
end

Procedure TRT(j, DT_min)
begin
    if (D^i_{j s^i_j} = DT_min)
    then ns ← s^i_j;
    else ns ← b | {b ∈ N_i and D^i_jb = DT_min};
    x ← j;
    while (D^i_{x ns} = Min{D^i_xb ∀ b ∈ N_i}
      and (D^i_{x ns} < ∞) and (tag^i_x = null))
    do x ← p^i_{x ns};
    if (p^i_{x ns} = i or tag^i_x = correct)
    then tag^i_j ← correct;
    else tag^i_j ← error;
    if (tag^i_j = correct)
    then begin
        if (D^i_j ≠ DT_min or p^i_j ≠ p^i_{j ns})
        then add (0, j, DT_min, p^i_{j ns})
            to LIST_i(x)  ∀x ∈ N_i;
        D^i_j ← DT_min; p^i_j ← p^i_{j ns};
        s^i_j ← ns;
    end
    else begin
        if (D^i_j < ∞)
        then add (0, j, ∞, null)
            to LIST_i(x)  ∀x ∈ N_i;
        D^i_j ← ∞; p^i_j ← null;
        s^i_j ← null;
    end
end
```

Figure 3.22: LPA Specification (cont...)

does not change x's information. Note that i does not have to wait for an update from c to infer that it should not use c as successor to j.

Blocking Temporary Loops

The example shown in Figure 3.24 illustrates the possibility of looping, even when path information is used. In the example, it is assumed that a has reported the implicit path aj to i and that b has reported the implicit path $bcdj$ to i. Furthermore, this path information

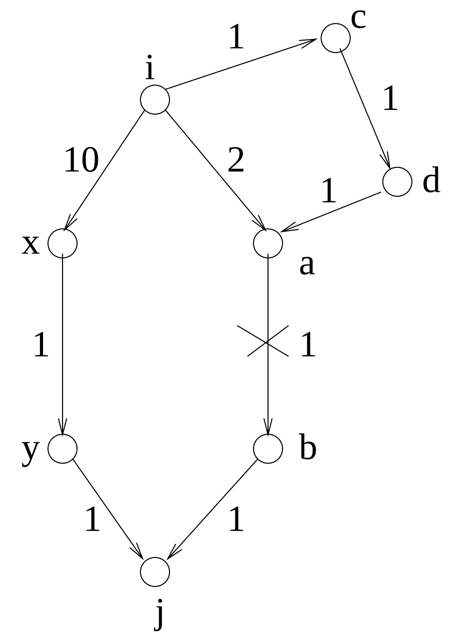

Figure 3.23: Updating Mechanism

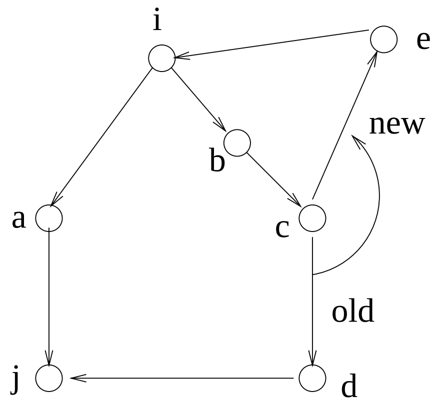

Figure 3.24: Possibility of a Loop

is outdated, because c has changed its successor from d to e and the new path information has not reached i. If link (i, a) failed, simply using path information would permit i to use b as successor to j. However, this would create a temporary routing loop.

To eliminate temporary loops, a router i forces its neighbors not to use it as a successor (next hop) when it detects the possibility of creating a temporary loop, before i changes its own successor. This is done using interneighbor synchronization mechanism based on the notion of feasible distance.

The feasible distance of router i for destination j (denoted by FD^i_j) is the smallest value achieved by its own distance to j since the last time i initialized itself or sent a query reporting an infinite distance to j. LPA allows a router to use a neighbor k as its successor to destination j only if it satisfies the following condition.

Feasibility Condition (FC): If at time t router i needs to update its current successor, it can choose as its new successor $s^i_j(t)$ any router $n \in N_i(t)$ such that $D^i_{jn}(t) + d_{in}(t) = D_{min}(t) = Min\{D^i_{jx}(t) + d_{ix}(t) | x \in N_i(t)\}$ and $D^i_{jn}(t) < FD^i_j(t)$. If no such neighbor exists and $D^i_j(t) < \infty$, router i must keep its current successor. If $D_{min}(t) = \infty$ then $s^i_j(t) = null$.

FC is used to establish an ordering of routers along a given loop-free path to j, i.e., all the routers in a loop-free path to j have feasible distances to j that decrease as j is approached. If router i does not find neighbor that satisfies FC, it is forced to send a query to its neighbors reporting an infinite distance to j and wait for the replies before it can change its own route. Because every router uses FC to decide whether to adapt a successor or to block paths through itself (described next), no temporary loops can exist.

Routing Table Updating

After procedure DT is executed, the way in which router i updates its routing table for a given destination depends on whether router i is *passive* or *active* for that destination. A router is *passive* if it has a *feasible successor*, or has determined that no such successor exists and is *active* if it is searching for a feasible successor. A feasible successor for router i with respect to destination j is a neighbor router that satisfies FC.

When router i is passive, it reports the current value of D^i_j in all its updates and replies. While router i is active, it sends an infinite distance in its replies and queries. An active router cannot send an update regarding the destination for which it is active; this is because any update sent during active state would necessarily have to report an infinite distance to ensure the correct operation of the inter-neighbor synchronization mechanism used in LPA.

If router i is passive when it processes an update for destination j, it determines whether or not it has a feasible successor, i.e., a neighbor router that satisfies FC.

If router i finds a feasible successor, it sets FD^i_j equal to the smaller of the updated value of D^i_j and the present value of FD^i_j. In addition, it updates its distance, predecessor, and successor making sure that only simple paths are used, as described in Section 2.4.

Router i then prepares an update message to its neighbors if its routing table entry changes. Alternatively, if router i finds no feasible successor, then it sets $FD^i_j = \infty$ and updates its distance and predecessor entries to reflect the information reported by its current successor. If $D^i_j(t) = \infty$, then $s^i_j(t) =$ null. Router i also sets the reply status flag ($r^i_{jk} = 1$) for all $k \in N_i$ and sends a query to all its neighbors. Router i is then said to be *active*, and cannot change its path information until it receives all the replies to its query.

The tagging mechanism used for routing table updating ensures that only those routing table entries affected by the input event will be traversed and updated. i.e., if there is a topology change in the existing routing tree, then only nodes which are downstream to that topological change need to be updated since the nodes upstream to the topological change will not be affected. This mechanism minimizes the processing that has to be done for each update message.

A path $P^i_{jk}(t)$ is defined by the predecessors reported by neighbor k to router j stored in i's distance table at time t. To ensure loop-free paths, path traversal from j back to k is made using the predecessor information. Complete or partial path can be traversed. Path traversal ends when either a predecessor x for which $tag^i_x = correct$ or $tag^i_x = error$ or neighbor k is reached. If $tag^i_x = error$, then tag^i_j is set to error also; otherwise, the neighbor k or a correct tag must be reached in which case tag^i_j is set to correct. This mechanism ensures loop-free paths without having to traverse the entire routing table.

Processing Queries and Replies

Queries and replies are processed in a manner similar to the processing of an update described above. If the input event that causes router i to become active is a query from its neighbor k, router i sends a reply to router k reporting an infinite distance. This is the case, because router k's query, by definition, reports the latest information from router k, and

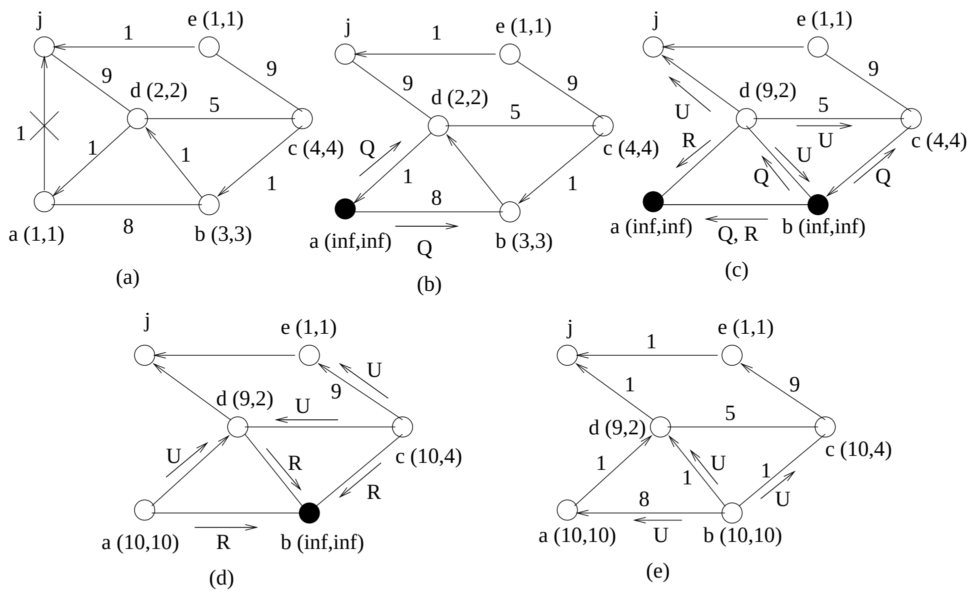

Figure 3.25: Example of LPA's Operation

router i will send an update to router k when it becomes passive if its distance is smaller than infinity. A link-cost change is treated as a number of updates.

Once router i is active for destination j, it may not have to do anything more regarding that destination after executing procedures RT and DT as a result of an input event. However, when router i is active and receives a reply from router k, it updates its distance table and resets the reply flag ($r^i_{jk} = 0$).

Router i becomes passive at time t when it receives replies from all its neighbors indicating that they have processed its query. As a result, router i is free to choose any neighbor that provides the shortest distance, if there is any. If such a neighbor is found, router i updates the routing table with the minimum distance as described for the passive state and sets $FD^i_j = D^i_j$.

A router does not wait indefinitely for replies from its neighbors because a router replies to all its queries regardless of its state. Thus, there is no possibility of deadlocks due to the inter-neighbor coordination mechanism.

If router i is passive and has already set its distance to infinity ($D^i_j = \infty$), and receives an input event that implies an infinite distance to j, then router i simply updates D^i_{jk} and

d_{ik} and sends a reply to router k with an infinite distance if the input event is a query from router k. This ensures that updates messages will stop when a destination becomes unreachable.

Figure 3.25 illustrates LPA's interneighbor coordination mechanism. The number adjacent to each link represents the weight of that link; U indicates updates, Q represents queries and R replies. The arrowhead from node x to node y indicates that node y is the successor of node x towards destination j; i.e., $s_j^x = y$. The label in the parenthesis assigned to node x indicates current distance (D_j^x) and the feasible distance from x to destination j (FD_j^x). Active nodes are indicated in black.

In the example (Figure 3.25), we assume that messages propagate across all links at the same speed, which is considered a *step*. Nodes process all messages received in the previous step in zero time.

When link (a, j) fails, node a updates its distance table by setting the distances from d and b to j equal to ∞, because the paths to j reported by both b and d include a. After that, node a is unable to find a feasible successor to j, because $D_{jb}^a = D_{jd}^a = \infty > 1 = FD_j^a$. Accordingly, it sends a query to all its neighbors (Figure 3.25(b)).

When node d receives a's query, it updates its distance table as follows: it sets $D_{ja}^d = \infty$ because a reports $D_j^a = \infty$, and it sets $D_{jb}^d = D_{jc}^d = \infty$, because the paths to j reported by b and d include node a. Because d uses a to reach j, d must also update its routing table. After updating its distance table, the neighbor that offers the shortest distance to j is j itself. Furthermore, $D_{jj}^d = 0 < 2 = FD_j^d$, and d sends an update to all its neighbor with $D_j^d = 1 + 9 = 10$ and a reply to node a (Figure 3.25(c)).

When node b receives a's query (before receiving a new update from d), it must set $D_{ja}^b = D_{jd}^b = D_{jc}^b = \infty$, because all its neighbors have reported a path to j that include node a. Because b's own path to j includes node a, it must update its routing table. Node b sends a query to its neighbors because every distance to j through any neighbor is infinity (Figure 3.25(c)).

When a receives the replies from d and b, it makes node d its new successor and also sends a reply to b (Figure 3.25(d)). When c receives the update from d and the query from b, it makes e its successor, because $D^c_{je} = 1 < 4 = FD^c_j$ and e offers the shortest path to j among all of c's neighbors. Accordingly, c sends an update with its new distance of 10 and a reply to b's query.

Finally, when b receives all the replies to its queries, it sets d as its successor and sends updates accordingly (Figure 3.25(e)).

Ensuring Simple Paths

Before updating the routing table, the algorithm ensures that all finite distances in the routing table corresponds to a simple path by allowing router i to select as the successors to destination only neighbors that satisfy the following property:

Property 1 *Router i sets $s^i_j = k$ at time t only if $D^i_{xk}(t) + d_{ik}(t) \leq D^i_{xp}(t) + l_{ip}(t)$ for every neighbor p other than k and for every node x in the path from i to j defined by the predecessors reported by neighbor k.*

Let $P^i_{jk}(t)$ denote the path from k to j defined by the predecessors reported by neighbor k to router i and stored in router i's distance table at time t. Procedure *TRT* enforces Property 1 by traversing all or part of $P^i_{jk}(t)$ from j back to k using the predecessor information. This path traversal ends when either a predecessor x is reached for which $tag^i_x = correct$ or *error*, or neighbor k is reached. If $tag^i_x = error$, then tag^i_j is set to *error* also; otherwise, the neighbor k or a correct tag must be reached, in which case tag^i_j is set to *correct*. Lemma 3.2 shows that this traversal correctly enforces Property 1, without having to traverse an entire implicit path; as the simulation results presented in Section 3.2.3 show, this makes LPA considerably more efficient than other prior path finding algorithms [CRKGLA89, Hum91].

Handling Topology Changes

When router i establishes a link with a neighbor k, it updates its link-costs table and assumes that router k has reported infinite distances to all destinations and has replied to any query for which router i is active; furthermore, if router k is a previously unknown destination, router i initializes the path information of router k and sends an update to the new neighbor k for each destination for which it has a finite distance. When router i is passive and detects that link (i,k) has failed, it sets $d_{ik} = \infty$, $D^i_{jk} = \infty$ and $p^i_{jk} = \text{null}$; after that, router i carries out the same steps used for the reception of a link-cost change message in passive state. When router i is active and loses connectivity with a neighbor k, it resets the reply flag and resets the path information i.e., assumes that the neighbor k sent a reply reporting an infinite distance.

It follows from this description of router i's operation that the order in which router i processes updates, queries and replies does not change with the establishment of new links or link failures. The addition or failure of a router is handled by its neighbors as if all the links connecting to that router were coming up or going down at the same time.

LPA is loop-free at every instant. This is possible by the single-hop interneighbor coordination and the efficient updating mechanism. The details of the proofs of loop-freedom of LPA are given in [GLAM96]. LPA has been shown to have a worst-case complexity of $O(x)$ after a single resource failure, where x is the number of routers affected in the topology change. This is also made possible by the efficient updating mechanism used to update the distance table entries in LPA.

3.2.2 Correctness of LPA

To prove that LPA converges to correct routing-table values in a finite time, we assume that there is a finite time T_c after which no more link-cost or topology changes occur.

Lemma 3.1 *LPA is live.*

Proof: Consider the case in which the network has a stable topology. When a router is in the active state and receives a query from a neighbor, the router replies to the query with an infinite distance. The router updates its distance table entries when either an update or a reply message is received in active state. On the other hand, when a router in passive state receives a query from its neighbor, it computes the feasible distance and updates its distance and routing tables accordingly. If the router finds a feasible successor, it replies to its neighbor's query with its current distance to the destination. If the router can find no feasible successor, it forwards the query to the rest of its neighbors and sends a reply with an infinite distance to the neighbor who originated the query. Accordingly, in a stable topology, a router that receives a query from a neighbor for any destination must answer with a reply within a finite time, which means that any router that sends a query in a stable topology must become passive after a finite time.

Consider now the case in which the network topology changes. When a link fails or is reestablished, an active router that detects the link status change simply assumes that the router at the other end of the link has reported an infinite distance and has replied to the ongoing query. Because an active router must detect the failure or establishment of a link within a finite time, and because router failures or additions are treated as multiple link failures or additions, it follows from the previous case that no router can be active for an indefinite period of time and hence the lemma is true. □

Lemma 3.2 *TRT correctly enforces Property 1.*

Proof: TRT correctly enforces Property 1 if the tag value given by TRT at router i for destination j equals correct. This is true only when the neighbor n that router i chooses as successor to j offers the smallest distance from i to each node in its reported implied path from n to j.

First note that, procedure DT is executed before TRT and ensures that router i sets $D^i_{jb} = \infty$ if its neighbor b reports a path to b that includes i. Therefore, TRT deals with simple paths only.

According to procedure TRT, there are two cases in which a router stops tracing the routing table (a) the trace reaches node i itself (i.e., $p^i_{xns} = i$), and (b) a node on the path to j is found with $tag^i_x =$ correct. We prove that the correct path information is reached in both cases.

Case 1: Assume that TRT is executed for destination j after an input event. The tag for each destination affected by the input event is set to null before procedure TRT is executed.

Therefore, if TRT is executed for destination j and node i (the source) is reached, the tag of each node in the path from i to j through neighbor n must be null. Therefore, the distance from i to j through n is the shortest path among all neighbors since node i chooses the minimum in row entry among its neighbors for a given destination j. The lemma is true for this case.

Case 2: If node x_1 with $tag^i_{x_1}$ = correct is reached, then it must be true that either node i or a node x_2 with $tag^i_{x_2}$ = correct is reached from x_1.

If node i is reached from x_1, then it follows from Case 1 that neighbor n offers the smallest distance among all of i's neighbors to each node in the implied subpath from n to x_1 reported by neighbor n. Furthermore, because x_1 is reached from j, node n must also offer the smallest distance among all of i's neighbors to each node in the implied subpath from x_1 to j reported by n. Therefore, it follows that the lemma is true if node i is reached from x_1 (from Case 1). Otherwise, if x_2 is reached, the argument used when i is reached from x_1 can be applied to x_2. Because router i always sets tag^i_i = correct and TRT deals with simple paths only, this argument can be applied recursively only for a maximum of $h < \infty$ times until i is reached, where h is the number of hops in the implicit path from n to j reported by n to i. Therefore, Case 2 must eventually reduce to Case 1 and it follows that the lemma is true. $\square$

Lemma 3.3 *The change in the cost or status of a link will be reflected in the distance and the routing tables of a router adjacent to the link within a finite time.*

Proof: Regardless of the state in which router i is for a given destination j, it updates its link-cost and distance table within a finite time after it is notified of an adjacent link changing its cost, failing, or starting up. On the other hand, router i is allowed to update its routing table for destination j only when it is in passive state for that destination. However, because LPA is live (Lemma 1), if router i is active for destination j, it must receive all the replies to its query regarding j within a finite time, i.e., when it becomes passive. When router i becomes passive for destination j, it executes Procedure TRT, which updates the routing-table entry for destination j using the most recent information in router i's distance table. This implies that any change in a link is reflected in the distance and routing tables of a neighbor router within a finite time T. $\Box$

Given Lemma 3.3 and our assumption about time T_c, a finite time must exist when all routers adjacent to the links that changed cost or status have updated their link cost and status information, and after which no more link-cost or topology changes occur. Let T denote that time, where $T_c \leq T < \infty$.

Theorem 1 *After a finite time $t \geq T$, the routing tables of all routers must define the final shortest path to each destination.*

Proof: Let $T(H)$ be the time at which all messages sent by routers with shortest paths having $H - 1$ hops ($H \geq 1$) to a given destination j have been processed by their neighbors.

Assume that destination j is reachable from every router.

For any router a adjacent to j, it follows from Lemma 2 that, if router a's shortest path to j is the link (a, j), then router a must update $D^a_j = d_{aj}$ by time $T = T(0)$ and the theorem is true for $H = 0$.

Because LPA is loop free at every instant (Theorem 1), the number of hops in any shortest path (as implied by the successor graph) is finite. Accordingly, the proof can proceed by induction on H.

Assume that the theorem is true for some $H > 0$. According to this inductive assumption, by time $T(H)$, router i must have a correct routing-table entry for every destination for which it has a shortest path of H hops or less. Property 1 must be satisfied for all such destinations and LPA enforces it correctly (Lemma 3.2). On the other hand, from the definition of $T(H + 1)$, it follows that any update messages sent by routers with shortest paths of H hops or less to j or any other destination have been processed by their neighbors by time $T(H + 1)$. Therefore, if router i's shortest path to destination j has $H + 1$ hops, Property 1 must be satisfied at router i for that destination by time $T(H + 1)$, because all possible predecessors for destination j must satisfy Property 1 at router i and that router must have the correct information for link (i, s^i_j) at time $T(0) < T(H+1)$ (Lemma 3.2). It follows that the theorem is true for the case of a connected network.

Consider the case in which j is not accessible to a connected component C of the network. Assume that there is a router $i \in C$ such that $D^i_j < \infty$ at some arbitrarily long time. If that is the case, j must satisfy Property 1 through at least one of router i's neighbors at that time; the same applies to such a neighbor, and to all the routers in at least one path from i to j defined by the routing tables of routers in C. This is not possible, because C is finite and LPA is always free of loops and live, which implies that, after a finite time $t_f \geq T$, all paths to j defined by the successor entries in the routing tables of routers in C must lead to routers that have set their distance to j equal to ∞. Therefore, because C is finite, LPA is live, and messages take a finite time to be transmitted, it follows that destination j will fail to satisfy Property 1 at each router within a finite time $t \geq t_f$, and routers must then set their distances to infinity, and the theorem is true. □

Theorem 2 *A finite time after t, no new update messages are being transmitted or processed by routers in G, and all entries in distance and routing tables are correct.*

Proof: After time T, the only way in which a router can send an update message is after processing an update message from a neighbor. Accordingly, the proof needs to consider three cases, namely: router i receives an update, a query, or a reply from a neighbor.

Consider an arbitrary router $i \in G$. Because LPA is live (Theorem 1) and router i obtains its shortest distance and corresponding path information for destination j in a finite time after T (Theorem 2), router i must be passive within a finite time $t_i \geq T$.

If router i receives an update for destination j from router k after time t_i, router i must execute Procedure Update. If router i has no path to destination j, D^i_j must be infinity and router k must report an infinite distance as well, because router i achieves its final shortest-path at time t_i; in this case, router i simply updates its distance table. On the other hand, if router i has a path to destination j, then $D^i_j < \infty$ and router i must find that FC is satisfied and execute Procedure TRT. Because an update entry is added only when the shortest distance or predecessor to j change, router i can send no update or query of its own.

If router i receives a query from a neighbor for destination j after time t_i, it must execute Procedure Query. If router i has no physical path to destination j, D^i_j must be infinity and router k must report an infinite distance in its query, because router i achieves its final shortest-path at time t_i; in this case, router i simply updates its distance table and sends a reply to router k with an infinite distance. On the other hand, if router i has a physical path to destination j, it must determine that FC is satisfied when it processes router k's query. Accordingly, it simply sends a reply to its neighbor with its current distance and predecessor to router j. Therefore, router i cannot send an update or query of its own when it processes a query from a neighbor after time t_i.

After time t_i, router i cannot receive a reply from a neighbor, unless it first sends a query after time t_i, which is impossible according to the above two paragraphs.

It follows from the above that, for any given destination, no router in G can generate a new update or query after it reaches its final shortest path and predecessor to that destination. Because every router must obtain its final shortest distance and predecessor to every destination within a finite time (Theorem 2), the theorem is true. □

3.2.3 Simulation Results

The performance of LPA was compared with that of DUAL and ILS. Simulation environment and instrumentation are as described in Section 3.1.2 and 3.1.3 respectively. Here, we focus our simulation study on Arpanet topology. We have studied the performance of the algorithm for single resource failures and recoveries and for random link cost changes. For random link-cost changes, links were chosen at random, with link costs chosen from the interval (0,1] and with a Poisson distributed interarrival time. Five independent runs were made and the average and the standard deviation of all quantities measured were determined. Interarrival time between the link-cost changes was varied to simulate multiple link-cost changes.

The link model allows the link delay to be set independently for each link. Each unit of time represents a step in which all currently available packets are processed. Although the

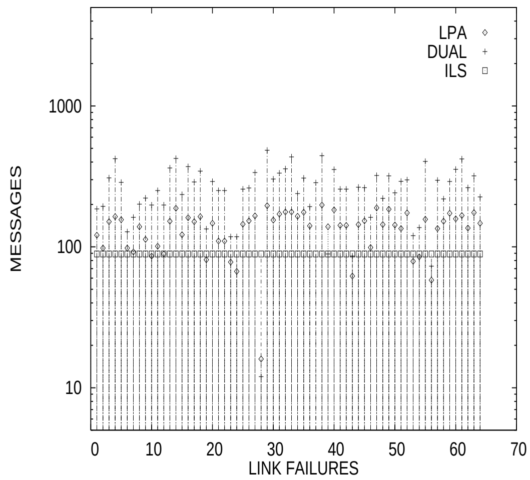

Figure 3.26: ARPANET Link Failure

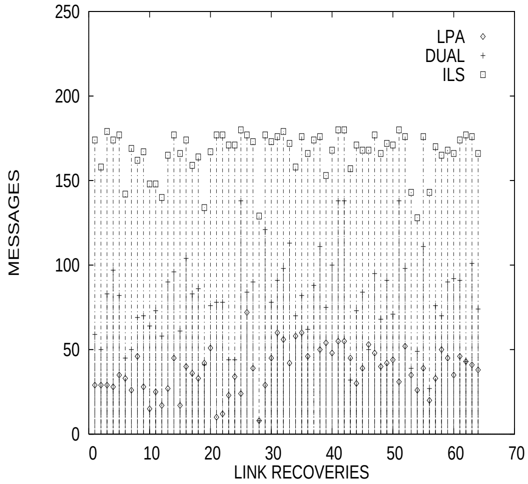

Figure 3.27: ARPANET Link Recovery

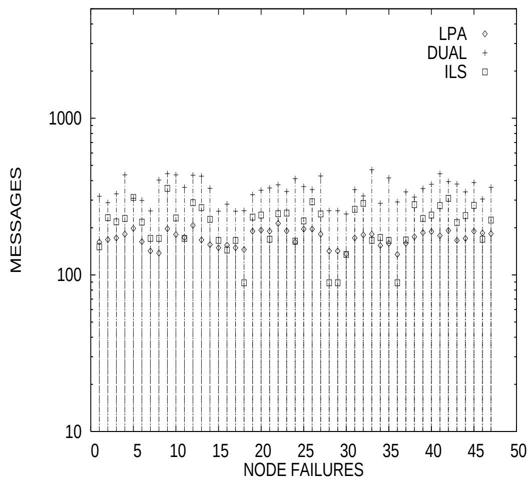

Figure 3.28: ARPANET Node Failure

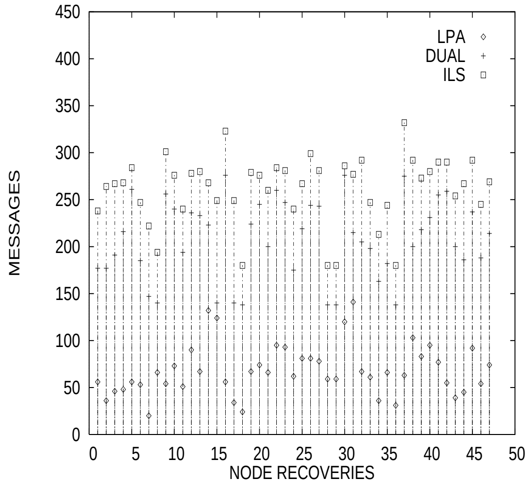

Figure 3.29: ARPANET Node Recovery

choice of input parameters causes the simulation to proceed synchronously, the node model treats each incoming packet asynchronously. Each input event is processed independently of other events received during the same simulation step.

The graphs in Figures 3.26 and 3.27 depicts the performance of the routing algorithms for a single topology change. The graphs show the number of messages exchanged before LPA, DUAL and ILS converge for every link failing and recovering in the Arpanet topology respectively. Similar graphs for each node failing and recovering are given in Figures 3.28 and 3.29, respectively. All topology changes were performed one at a time and the algorithms were allowed to converge after each such change before the next resource change

Table 3.1: Routing Algorithm Response to a Change in Link Cost

Parameter	LPA		DUAL		ILS	
	mean	sdev	mean	sdev	mean	sdev
Los-Nettos Cases						
Event Count	21.4± 0.47	35.6± 2.8	34.0± 0.24	45.0± 0.54	18.5± 0.05	18.5± 0.04
Packet Count	12.2±0.15	7.5± 0.41	15.45± 0.14	18.6± 0.25	17.5± 0.05	17.5± 0.04
Duration	3.95± 0.06	1.76± 0.07	4.9±0.06	2.17± 0.07	4.06± 0.01	0.46± 0.03
Operation Count	45.7± 0.23	23.1± 1.1.7	44.0± 0.24	52.9± 0.5	473.9± 3.08	475.5± 3.09
NSFNET Cases						
Event Count	34.65± 0.83	72.86± 2.7	50.97± 1.2	68.7± 2.5	28.5± 0.09	28.5± 0.09
Packet Count	14.97± 0.15	14.78± 0.47	21.81± 0.54	27.1± 0.82	27.5± 0.1	27.5± 0.09
Duration	4.624± 0.05	2.45± 0.05	5.516± 0.17	2.57± 0.15	4.7± 0.02	0.46± 0.01
Operation Count	55.323± 0.42	42.64± 1.21	63.97± 1.2	78.8± 2.35	1012.88± 3.9	1014.5± 3.76
ARPANET Cases						
Event Count	247.5± 16.09	679.7± 24.6	350.09± 15.08	501.4± 22.7	84.1± 0.23	84.6± 0.21
Packet Count	55.8± 1.86	66.8± 3.0	81.8± 2.99	102.9± 4.9	83.1± 0.23	83.6± 0.21
Duration	7.042± 0.22	4.9± 0.09	10.84± 0.45	4.75± 0.55	7.74± 0.028	0.74± 0.022
Operation Count	269.7± 8.05	359.2± 12.2	396.1± 15.1	534.6± 22.6	13613.1± 51.0	13691.3± 47.4

occurs. The ordinates of the graphs represent the identifiers of the links and the nodes, while the data points show the number of messages exchanged after each resource change in each of these figures.

The response of the algorithms for a single link-cost change is given in Table 3.1. The table gives the average value and the standard deviation along with the statistical error for each link-cost change. The statistical errors were determined based on repeated trails. For a single link-cost change, LPA is faster and needs fewer messages and operations than both DUAL and ILS in all topologies. We can conclude from these results that LPA has a better average performance than ILS and DUAL after any single link-cost or topology change.

Dynamic Response to a Single Change

To study the dynamic behavior of the routing algorithms, we ran an exhaustive series of test cases for all the node and the link failures and recoveries and recorded the message related statistics. A statistical characterization of the performance of the routing algorithms was obtained by treating every node change as a separate case and by computing a distribution as a function of time. In this section, we present the results of the dynamic behavior of the

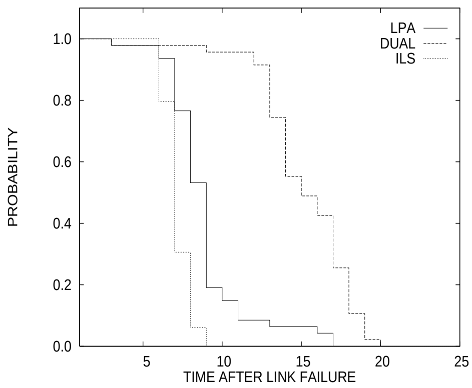

Figure 3.30: Probability of packets in transit for Link Failure

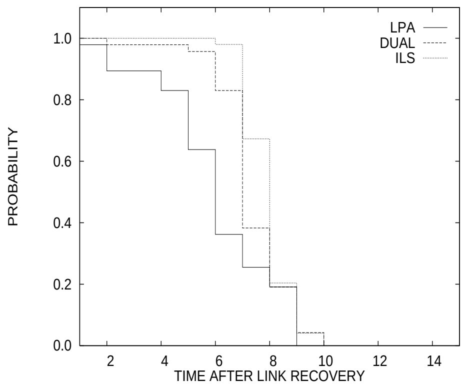

Figure 3.31: Probability of packets in transit for Link Recovery

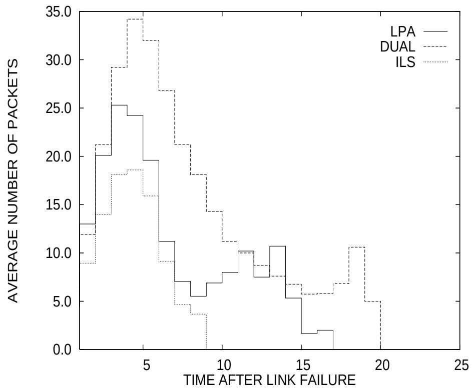

Figure 3.32: Average number of packets for Link Failure

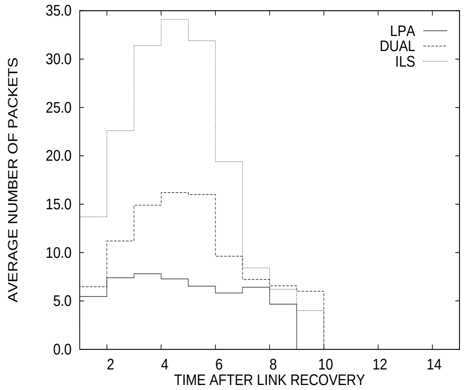

Figure 3.33: Average number of packets for Link Recovery

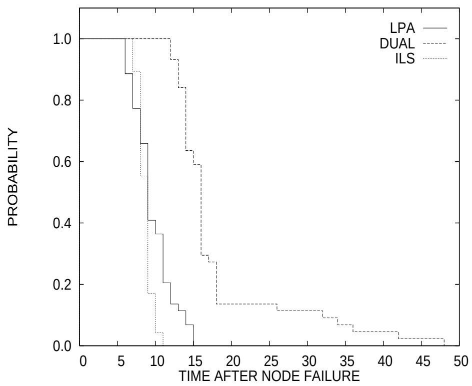

Figure 3.34: Probability of packets in transit for Node Failure

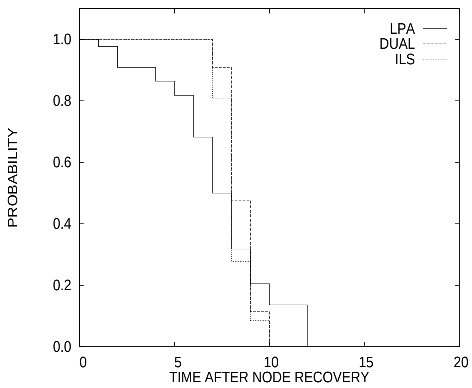

Figure 3.35: Probability of packets in transit for Node Recovery

routing algorithms for the *Arpanet* topology. In the instrumentation, we do not consider paths from a node to itself, because they do not require a network. In our simulations, we have taken care to handle these cases.

Figures 3.30–3.37 show the transient response of the routing algorithms (probability of packets in transit and the average number of messages that are exchanged) after a link failure, link recovery, node failure and node recovery respectively. These are shown as a function of time.

The results indicate that for a link failure, ILS performs better than DUAL and LPA in terms of the number of messages exchanged. LPA's average performance is much better than ILS for resource addition at any time after the change. The performance of LPA is comparable to ILS after a node failure. In all cases, LPA outperforms DUAL. The probability of packets being in transit for LPA after a resource recovery is less than ILS and DUAL. The average packet length for LPA after a resource change is much smaller than DUAL. This is because of the single hop interneighbor coordination mechanism and the tagging mechanism used in LPA.

Response to Multiple Link-Cost Changes

The steady-state behavior of the algorithms is more interesting with multiple link-cost changes than the transient response after each topology change. Figures 3.38–3.41 shows the average number of update messages when such messages are in transit, the average lengths of messages, the average number of messages in transit and the probability that the messages are in transit as a function of the interarrival times between link-cost changes for LPA, DUAL and ILS. This again is for the Arpanet topology. From [ZGLA92], it has been observed that the behavior of DUAL and ILS for multiple link-cost changes is similar for different network topologies; our conjecture is that the same is true for LPA.

For very long interarrival times, the number of messages during busy periods is independent of the interarrival time because the probability of two topology changes occurring simultaneously is small. In this case, the performance approaches that of single link-cost change. When the interarrival time approaches the network diameter, this situation changes and the number of messages during the busy period increases because of multiple topology changes occurring simultaneously.

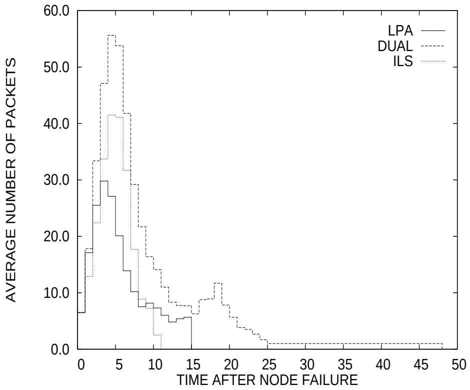

Figure 3.36: Average number of packets for Node Failure

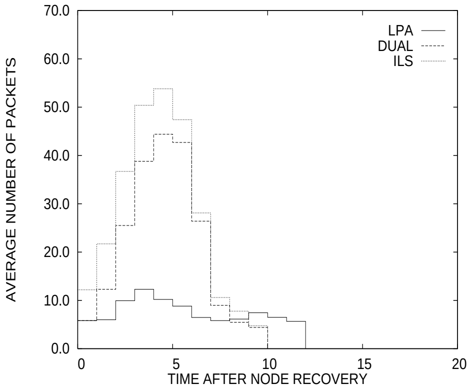

Figure 3.37: Average number of packets for Node Recovery

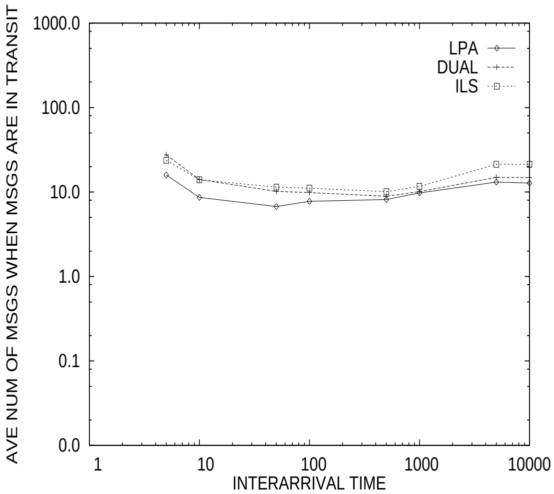

Figure 3.38: Average Number of Messages when messages are in Transit

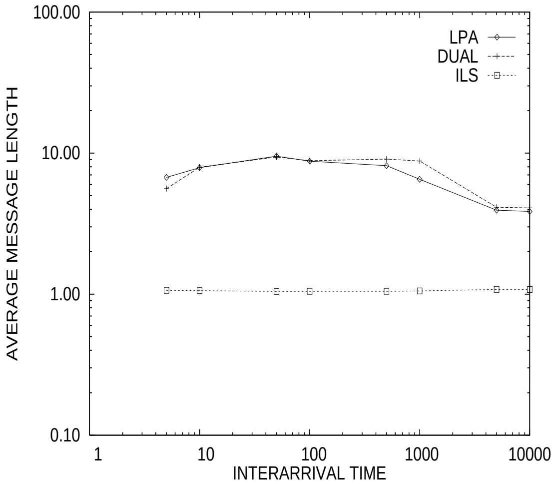

Figure 3.39: Average Message Length

The average number of messages exchanged when messages are in transit is slightly less for LPA compared to DUAL and ILS. This again is because of the single-hop internodal synchronization mechanism and updating of the distance-table entries, which has been explained earlier. All curves are of roughly the same shape. For ILS, the average message length is close to 1. DUAL and LPA have longer messages as a message can contain multiple updates; this occurs when messages from the routers at both ends of a link that changes cost arrive at some router at the same time. With the increase in the interarrival time between changes, the average number of messages drops down as it now approaches a single link-cost change. The average number of messages that are in transit and the probability that messages are in transit are significantly smaller for LPA compared to DUAL and ILS.

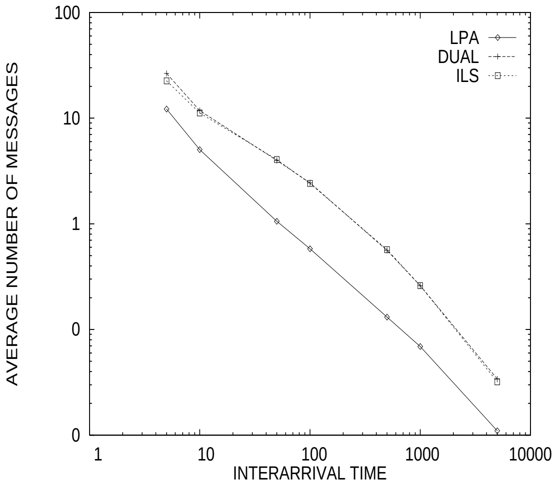

Figure 3.40: Average Number of Messages in Transit

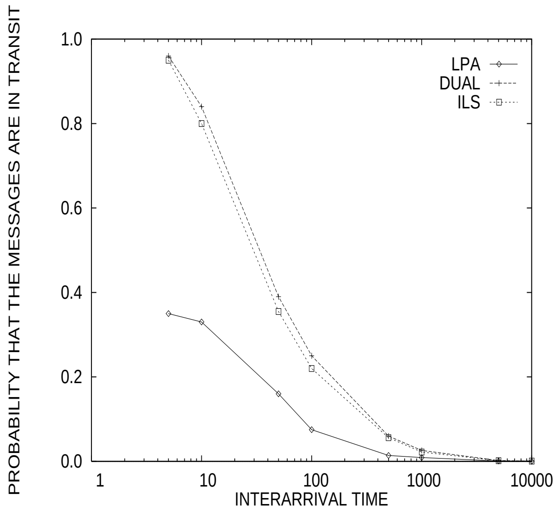

Figure 3.41: Probability that Update Messages are in Transit

However, DUAL and ILS roughly follow the same curve. The results clearly indicate that LPA incurs smaller overhead traffic than either DUAL and ILS when multiple link-cost changes occur.

3.3 Summary

In this chapter, we presented two path-finding algorithms, PFA and LPA, for loop-free routing in flat networks. Both the algorithms are based on the principle of deriving implicit path information from the predecessor node entries in the distance and routing tables at each router. PFA reduces the number of cases a temporary routing loop can occur while LPA eliminates the formation of temporary routing loops without internodal synchronization spanning multiple hops or the communication of complete or variable length path information. We also have presented the performance of these two algorithms and have compared with the state of the art routing techniques namely, DUAL and ILS.

The statistical techniques used in our analysis provide a way of characterizing the performance of various algorithms, and can be used as a basis for a tradeoff analysis during network design. Our study shows that both PFA and LPA are more efficient than the algorithms compared. The time behavior of loop-free distance-vector algorithms show that the parameters such as packet length can change as updates propagate, thereby suggesting the possibility of heuristics that can exploit local conditions. Our analysis indicates that overall, LPA and PFA perform significantly better than DUAL and ILS.

In the following chapters, we propose several routing protocols for different networking environments, which use PFA or LPA as the basic routing algorithm.

Chapter 4
A Hierarchical Routing Algorithm

The loop-free path-finding algorithm (LPA) has been shown to maintain loop-free routing tables. LPA obtains correct routing tables after topological and link-cost changes faster and with less processing and communication overhead than link-state algorithms and prior loop-free routing algorithms based on vectors of distances. The limitation of LPA and prior routing algorithms based on routing trees is that it requires the routers to maintain more "host routes" than would be needed in a traditional distance-vector algorithm.

Routing information maintained at each router has to be updated frequently to adapt to changes in the topology and congestion of the internetwork. In an internetwork with a flat routing structure, the size of the routing tables grow linearly with the number of destinations in the network. Due to this, the routing information that is required to be maintained at a node may become excessive in terms of storage and CPU utilization. The information exchanged among nodes may prove to be expensive in terms of channel bandwidth since updates need to be exchanged frequently in order to maintain up to date network state information. Accordingly, aggregation of routing information becomes a necessity in any type of routing protocol.

For routing in large networks, the aggregation of routing information is achieved through a hierarchical partitioning of the network. The main idea of hierarchical routing is to maintain exact routing information regarding nodes very close to it and less detailed information regarding nodes that are farther away from it. The goal of maintaining hierarchy of information is to reduce the size of the routing database maintained at each router so that the exchange of topology information among the routers can be minimized. The objective of doing so is to obtain a reasonable compromise among the size of routing tables, number of updates required to maintain such tables and the speed with which updates are propagated.

In this chapter, we present a hierarchical routing algorithm which we call *Hierarchical Information Path-based Routing* (HIPR), which can be used in large systems.

4.1 Prior Work

Hierarchical routing for datagram computer networks was first proposed by McQuillan [McQ74] to reduce the overhead by limiting the amount of routing information each routing node maintains and to reduce the length of the routing table. In order to reduce the amount of information maintained at each node and the communication overhead, the network is organized into clusters or areas by grouping together (clustering) the nodes which are close by. Each of these clusters is a single addressable entity from the point of view of higher level clusters. The topology within the area is transparent to the nodes outside the area. The distance from a source node to an area represents the actual distance in physical hops (number of nodes traversed) from the source node to the destination remote node. It then becomes important to determine a specific clustering structure, cluster size, and the number of clusters that will result in a minimum table length. Given that an optimal clustering technique has been determined, an efficient routing algorithm can be applied to these clusters.

McQuillan [McQ74] proposed an extension to the old Arpanet routing algorithm for hierarchical networks. This scheme was analyzed by Kamoun and Kleinrock [Kam76]. According to McQuillan, the nodes in the network are organized into m levels of clusters, with each node belonging to exactly one cluster at each level of the hierarchy. Nodes represent clusters at level 0; a group of nodes form a cluster at level 1, called 1-cluster; Similarly, a group of $(k-1)$ clusters forms a cluster at level k, called a k-cluster. The distance from a source node to a k cluster represents the length of the shortest-path from the source node to another node in the k-cluster. Each node maintains a routing table with distance entries to all other nodes in its 1-cluster, as well as to other i-cluster nodes that belongs to its $(i+1)$ cluster.

There have been several subsequent proposals for hierarchical routing which vary in the way in which the nodes are organized (addressing scheme) and the routing algorithms used. Baratz and Jaffe [BJ83] have proposed a modification to McQuillan's hierarchical routing scheme to support *virtual-circuit* routing in a two-level hierarchical network. In this scheme, each node maintains an entry in its routing table for every other node in its same l-cluster as well as for each boundary node in the network; a collaboration protocol is then used to obtain virtual circuit to optimal lengths between nodes in different clusters.

Meanwhile, Ramamoorthy and Tsai [RT83] have extended the new Arpanet routing algorithm [MRR80] to work in a hierarchical network. In their scheme, nodes within the same 1-cluster use intra-cluster updates, according to the *shortest-path first* (SPF) algorithm, to obtain the shortest paths to one another. This implies that each node maintains a full topology of its 1-cluster in addition to its intra-cluster routing table, and that every update message is distributed to all the nodes in the 1-cluster. Boundary or border nodes are in charge of inter-cluster routing updates. A virtual network at level i is defined as one in which a link between two i-level boundary nodes is formed by the shortest path between them; the delay of the virtual link is then the delay in the shortest path. The SPF algorithm is used at every level of the hierarchy to allow each i-level boundary node to compute the shortest path to another i-level boundary node.

Besides these algorithms, as part of the DARPA-sponsored SURAN project, a number of hierarchical network architectures have been proposed and analyzed for large packet radio networks [GLAS85, Lau86]. In these schemes, a modification of McQuillan's scheme has been used. The algorithms for computing the shortest paths include variations of DBF. Finally, Seeger and Khanna employ Ramamoorthy and Tsai's hierarchical scheme in a two-level hierarchy designed to reduce routing overhead in the DDN [SK86].

OSPF [Moy94], a link-state protocol, implements an area-based routing architecture similar to the above, and divides the Internet into areas connected by boundary nodes linked by a physical backbone. Routing nodes connected to the backbone serve the role of boundary nodes called *backbone nodes* or *border nodes.* Routers not connected to the backbone correspond to simple nodes called *area nodes.* Boundary nodes exchange information about the backbone links adjacent to them, and area nodes exchange information about their adjacent intra-area links.

A boundary node reports to other boundary nodes the distance to multiple destinations in its own area. Topology information about an area is not directly propagated to routers outside the area. OSPF allows entries in one area to be represented in other areas and in the backbone as a cost associated with a *summary address* and a *mask.* To route to a destination address, the mask is applied to the address; if it matches the summary address, the routing-table entry for that summary is used. If there are multiple matches, the most specific is used.

OSPF provides a means to transfer and maintain a set of tables giving the costs of links and a cost for summary descriptions of distant nodes in the networks. Each boundary node computes a summary distance for mask/address combinations that match the nodes in each area attached to the backbone, and sends these distances to other boundary nodes. The boundary nodes then compute the shortest path to each globally visible node and summary entry, treating the summary as a link. Boundary nodes also send distances of globally visible nodes and summarizes outside an area to nodes in that area. The local simple nodes then treat such a distance as the cost of a link from the boundary node to that destination. Shortest paths are computed using Dijkstra's shortest-path algorithm for simple nodes and the boundary nodes.

With very few exceptions [GLAZ94], prior proposals to hierarchical routing have assumed variants of DBF or topology-broadcast algorithms. In the following sections we present, verify and analyze the performance of the first hierarchical routing algorithm (HIPR) based on the maintenance and exchange of hierarchical routing trees. The main idea of HIPR consists of providing a distributed implementation of Dijkstra's shortest path algorithm running over a hierarchical graph organized in areas according to McQuillan's scheme for hierarchical routing. LPA is extended for this purpose. HIPR constitutes the basis for a new Internet routing protocol that are as simple as RIPv2 [Mal94].

4.2 Network Model

Following the McQuillan's approach to area-based routing, hosts and networks can be organized into L levels of areas. An area at level k is called a k-area. Each router or destination belongs to only one area at each level of the hierarchy above level-1 and each k-area is a proper subset of only one $(k+1)$-area. For simplicity, no further mention is made of the destinations directly attached to routers through LANs or point-to-point links. Hence for the purposes of discussion, a destination is a router or a k-area.

Routers are themselves 0-areas; a group of nodes forms a 1-area and a group of k-areas forms $(k+1)$-area. A *boundary node* or a *border node* is defined as a router with direct connectivity with its peer boundary nodes in areas to which it does not belong. A k-level border node is the border node that connects k-areas. The distance from a source router to a remote k area represents the length of the true shortest-path from a node to any remote area. Clearly, the distance from a router to another router in the same 1-area is the true shortest path distance. Similarly, the distance from a router to itself (or any directly adjacent host) is 0 and the distance from the router to its own k-area is 0.

In OSPF terminology, a routing node connected to the backbone network serves as a boundary node. Routers not connected to the backbone are simple nodes. How the areas of the network are chosen is outside the scope of this paper, just as the way in which areas are chosen is not part of the OSPF specification. For example, by taking into account destination behind routers, masks and subnets, our description of the network hierarchy can be mapped into a hierarchical addressing scheme that would be practical on an Internet. For our purposes, it suffices to illustrate the fact, for HIPR, a destination is a single entity or an arbitrary aggregation of entities following McQuillan's scheme.

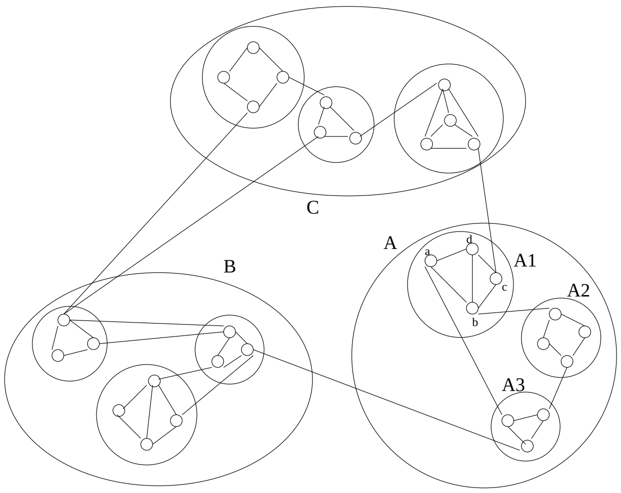

Figure 4.1: Example of the Hierarchical Network Topology

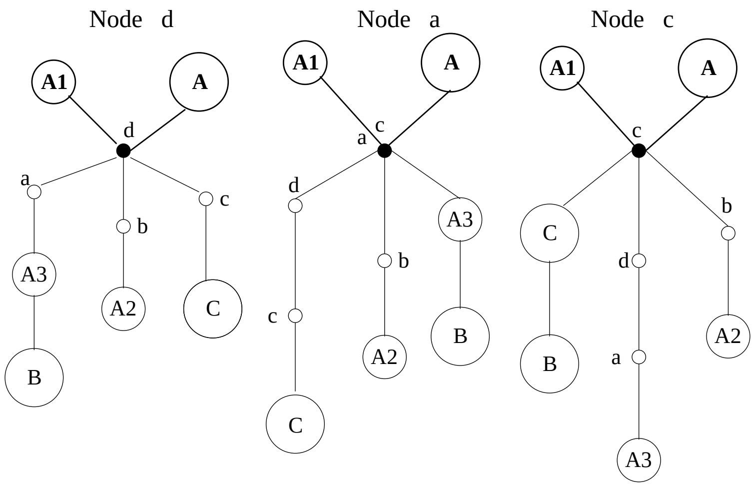

Figure 4.2: Hierarchical Routing Trees at Nodes

4.3 HIPR

4.3.1 Design Principle

The basic design concept of HIPR is simple. Each router communicates to its neighbors its hierarchical routing tree in an incremental fashion. Its hierarchical routing tree consists of all its preferred hierarchical shortest paths to each known individual destinations in its own 1-area, all known highest-level areas, and in general all $(k-1)$-areas within its own k-area. This means that border nodes forbid detailed routing tree information regarding their own areas from percolating to other areas. Hence, for a hierarchical routing tree an

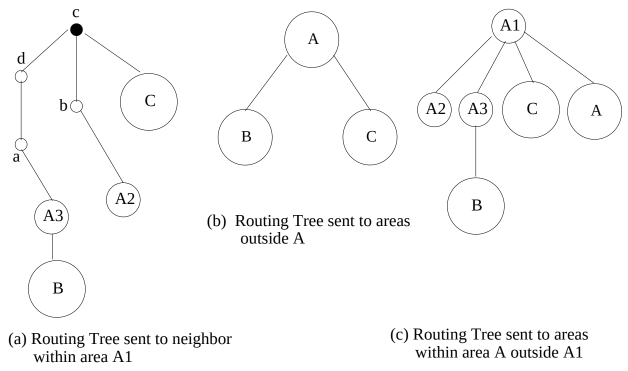

Figure 4.3: Hierarchical Routing Trees Sent by Border Nodes

entire foreign area is simply another node in the tree. Figures 4.1 and 4.2 illustrates this idea. Figure 4.1 shows the topology of a network organized hierarchically into three levels, and Figure 4.2 shows the hierarchical routing tree that router d needs to communicate to its neighbor routers (within area $A1$) and the hierarchical routing trees of its two neighbors a and c. Bold nodes in the trees correspond to "self reference" entries in the router's routing table. Notice that the path from d to remote areas consists of both routers within router d's 1-area and other remote areas.

Consider a border node in the given topology (say c). The hierarchical routing tree propagated to all nodes within the area containing node c is shown in Figure 4.3(a). For the areas outside the area containing node c (i.e., outside A), the hierarchical routing tree sent to the peer nodes of other areas is shown in Figure 4.3(b).

Consider border node b in Figure 4.1. This is a border node connecting areas $A1$ and $A2$ which are contained in the bigger area A. The routing tree sent by node b to its peer border node in area $A2$ (within area A) is shown in Figure 4.3(c).

Routers exchange their hierarchical routing trees incrementally by communicating only the hierarchical distance and second-to-last hop (predecessor) to each destination. In the case of destinations within router d's 1-area, the second-to-last hop consists of a router. In the case of a remote area known to router d, the predecessor consists of either a border node in router d's 1-area or a remote area. The hierarchical distance to a destination consists of the true shortest distance to a border node. The rest of this section describes the information and procedures that HIPR uses to update the hierarchical routing trees of routers distributedly, examining that no routing table loops are ever formed. In essence, HIPR implements Dijkstra's shortest-path algorithm distributedly over a hierarchical graph.

Although HIPR supports multiple levels of areas, OSPF can support only two levels (subnets and backbone). In this paper, we discuss HIPR based on only one level of areas to simplify its description and because supporting a minimum of one level of areas is critical for using HIPR as part of an Internet routing protocol, given that, for scalability purposes, Internet routing protocols maintain routing information about networks (not individual hosts) and must support subnetting.

4.3.2 Information Maintained at a Router

Each router maintains a single routing table that, for the purpose of the description, can be thought of as having two parts: a *node-level routing table* (NRT) and a *area level routing table* (ART). The NRT portion of the routing table maintains information about the routers and destinations in the same area with which a node is affiliated. The ART portion of the routing table maintains information about other areas (higher-level areas to which a node belongs). Both parts of the routing table are updated using the same algorithm.

The entry for destination j in node i's NRT consists of the destination's identifier, the distance to the destination (D^i_j), the successor (s^i_j), the predecessor (p^i_j) along the preferred path (shortest path) to the destination, and the feasible distance to the destination (FD^i_j), which we define subsequently. The NRT also maintains a marker (denoted by tag^i_j) used to update the routing table entries. For destination j, tag^i_j specifies whether the entry corresponds to a simple path (tag^i_j = correct), a loop (tag^i_j = error) or a destination that has not been marked (tag^i_j = null). This marker is used to reduce the number of routing table entries that need to be processed after each input event impacting the routing table. ART maintains similar information for each area known to node i.

The distance from a source to destination area (AD^i_j) is the length of the hierarchical path from source to the destination area. There can be two variations as to how the distance information is maintained. It can be either the hop count to reach a boundary node of the destination area or it can be the actual distance to reach the boundary node. Hop count just gives the number of hops required to reach the destination area and does not necessarily correspond to the actual path length and hence it need not be the shortest path. In our simulations we have assumed hop counts for simplicity.

The successor entry for the ART is the *next area* in the path towards the destination area (As_j^i) and the predecessor entry (Ap_j^i) is the *area* previous to the destination area. The successor and the predecessor entries will be *null* when i determines that the individual or area destination is unreachable. The ART also maintains a marker ($Atag_j^i$) for each known area.

From the above, it is clear that HIPR simply maintains a hierarchical routing tree in a concise manner.

In addition to its routing table, a router also maintains a *distance table*, *link-cost table* and a *reply-status table*. Distance table maintained at each router is a matrix containing for each destination j, the hierarchical routing tree reported by its neighbors $k \in N_i$. At router i, D_{jk}^i and p_{jk}^i represents the distance and the predecessor reported by neighbor k for destination j respectively.

The link-cost table contains the cost of each link adjacent to the node maintaining the table. The cost of the link is denoted by l_{ik}. The cost of an inactive link is set to infinity.

The reply-status table at node i maintains the values of the reply status flag (r_{jk}^i) for each neighbor, k, and for each known destination and area, j, to node i. Each entry in this table indicates whether or not a node is waiting to get a reply from its neighbor in response to its query.

4.3.3 Information Exchanged between Nodes

Routing information is exchanged among neighboring nodes by means of update messages. An update message from router i consists of a vector of entries reporting incremental updates of its routing table; each entry specifies an update flag (denoted by u_j^i), a destination j (i.e., an individual node or an area), the reported distance to that destination, and the reported predecessor (individual node or an area) in the path to the destination. The update flag indicates whether the entry is an update ($u^i_j = 0$), a query ($u^i_j = 1$) or a reply ($u^i_j = 2$). The distance in a query is always set to ∞.

Because every router reports to its neighbors the second-to-last hop in the shortest path to the destination, the complete path to any destination (called the implicit path to the destination) is known by the router's neighbors. This is done by the path traversal on the predecessor entries reported by the router.

In the specification of HIPR, the successor to destination j for any router is simply referred to as the successor of the router. Same reference applies to other information maintained by the router. Similarly, updates, queries and replies refer to destination j unless otherwise stated.

Figures 4.4 and 4.5 specify HIPR. The procedures used for initialization are *Init1* and *Init2*; procedure *Message* is invoked when a router processed a message. Procedures *linkup*, *linkdown* and *linkchange*, which are referred to as event-handling procedures, are executed when a router detects a new link, the failure of a link or the change in the cost of a link. Procedure *Message* calls *Update*, *Query* or *Reply* to handle an update, a query or a reply respectively. An important characteristic of all event-handling procedures is that they mark tag^i_j = null for each destination j affected by the input event.

The information propagated to the neighbor depends on whether the node is a border node or not. Border nodes block any information about its local area before sending an update to other border nodes (in its peer areas). This ensures that the algorithm is scalable.

Router i initializes itself in passive state with an infinite distance for all its known neighbors and with a zero distance to itself. After initialization, router i sends updates containing the distance to itself to all its neighbors.

4.3.4 Distance Table Updating

The procedures used in HIPR to update the entries of the distance tables are similar to the procedures used in LPA [GLAM95]. The key difference is that, as we have stated, a border

```
Procedure Init1
when router i initializes itself
do begin
    set a link-state table with
    costs of adjacent links;
    N ← {i}; N_i ← {x | d_ix < ∞};
    initialize A_i;
    for each (x ∈ N_i)
    do begin
        N ← N ∪ x; tag^i_x ← null;
        s^i_x ← null; p^i_x ← null;
        D^i_x ← ∞; FD^i_x ← ∞
    end
    for each (y ∈ A_i)
    do begin
        Atag^i_y ← null; AD^i_y ← ∞;
        Ap^i_y ← null; Ap^i_y ← null;
    end
    s^i_i ← i; p^i_i ← i; tag^i_i ← correct;
    D^i_i ← 0; FD^i_i ← 0;
    for each j ∈ N call Init2(x, j);
    for each (n ∈ N_i) do
    add (0, i, 0, i) to LIST_i(n);
    for each (n ∈ A_i) do
    add (0, i, 0, i) to LIST_i(n);
    call Send
end

Procedure Query(j, k)
begin
    if (r^i_jx = 0∀x ∈ N_i)
    then begin
        if (D^i_j = ∞ and D^i_jk = ∞)
        then add (2, j, D^i_j, p^i_j)
          to LIST_i(k)
        else begin
           call Passive_Update(j);
           add (2, j, D^i_j, p^i_j)
             to LIST_i(k);
        end
    else call AU(j, k)
end

Procedure Update(j, k)
begin
    if (r^i_jx = 0, ∀x ∈ N_i)
    then begin
          if ((s^i_j = k) or (D^i_jk < D^i_j))
          then call Passive_Update(j)
    end
    else call AU(j, k)
end

Procedure Init2(x, j)
begin
    D^i_jx ← ∞; p^i_jx ← null;
    s^i_jx ← null; r^i_jx ← 0;
end

Procedure Message
when router i receives a message
  on link (i, k)
begin
    for each entry (u^k_j, j, RD^k_j, rp^k_j)
      such that j ≠ i
    do begin
       if (j ∉ N)
       then begin
          if (RD^k_j = ∞)
          then delete entry
          else begin
             N ← N ∪ {j}; FD^i_j = ∞;
             for each x ∈ N_i
               call Init2(x, j)
             tag^i_j ← null;
             call DT_Update(j, k)
          end
       end
       else
          tag^i_j ← null; call DT_Update(j, k)
    end
    for each entry (u^k_j, j, RD^k_j, rp^k_j) left
    such that j ≠ i
    do case of value of u^i_j
       0: [Entry is an update]
          call Update(j, k)
       1: [Entry is a query]
          call Query(j, k)
       2: [Entry is a reply]
          call Reply(j, k)
    end
    call Send
end

Procedure Reply(j, k)
begin
    r^i_jk ← 0;
    if (r^i_jn = 0, ∀n ∈ N_i)
    then if ((∃x ∈ N_i | D^i_jx < ∞)
      or (D^i_j < ∞))
       then call Passive_Update(j)
    else call Active_Update(j, k)
end

Procedure Send
begin
    if (i is not a border node)
    begin
        for each (n ∈ (N_i ∪ A_i))
        do begin
            if (LIST_i(n) is not empty)
            then send message with
              LIST_i(n) to n
            empty LIST_i(n)
        end
    end
    else begin
        for each (n ∈ A_i)
        do begin
            if (LIST_i(n) is not empty)
            then send message with
              LIST_i(n) to n
            empty LIST_i(n)
        end
    end
end

Procedure Passive_Update(j)
begin
    DT_min ← Min{D^i_jx ∀ x ∈ N_i};
    FCSET ← {n | n ∈ N_i, D^i_jn = DT_min,
      D^n_j < FD^i_j};
    if (FCSET ≠ ∅) then begin
      call RT_Update(j, DT_min);
      FD^i_j ← Min{D^i_j, FD^i_j}
    end
    else begin
       FD^i_j = ∞; r^i_jx = 1,  ∀x ∈ N_i;
       D^i_j = D^i_{j s^i_j};
       p^i_j = p^i_{j s^i_j};
       if (D^i_j = ∞) then s^i_j ← null;
       ∀ x ∈ N_i do begin
         if (query and x = k)
         then r^i_jk ← 0;
         else add (1, j, ∞, null)
           to LIST_i(x)
       end
    end
end
```

Figure 4.4: HIPR Specification

node at level k running HIPR supports hierarchical routing by making sure that no routing information regarding destinations in its own 1-area or any $(k-1)$-area in its own k-area percolates to a neighbor border node in another k-area.

When router i receives an input event regarding neighbor k indicating the change in the link cost (i, k), it updates its link-cost table with the new cost of the link and then updates the distance table entries making sure that the implicit paths after the changed state of the network does not imply any loops. Updating the distance table entries erases the outdated path information by making the path information consistent with the latest update. This is

```
Procedure Link_Up (i, k, d_ik)
when link (i, k) comes up do begin
    d_ik ← cost of new link;
    if (k ∉ N) then begin
        N ← N ∪ {k}; tag^i_k ← null;
        D^i_k ← ∞; FD^i_k ← ∞;
        p^i_k ← null; s^i_k ← null;
        for each x ∈ N_i do call Init2(x, k)
    end
    N_i ← N_i ∪ {k};
    for each j ∈ N do call Init2(k, j);
    for each j ∈ N − k | D^i_j < ∞ do
        add (0, j, D^i_j, p^i_j) to LIST_i(k);
    call Send
end

Procedure Link_Down(i, k)
when link (i, k) fails do begin
    d_ik ← ∞;
    for each j ∈ N do begin
        call DT_Update(j, k);
        if (k = s^i_j) then tag^i_j ← null
    end
    delete column for k in distance table;
    N_i ← N_i − {k};
    delete r^i_jk;
    for each j ∈ (N − i) | k = s^i_j
        call Update(j, k)
    call Send
end

Procedure Active_Update(j, k)
begin
    if (k = s^i_j) then begin
        D^i_j ← D^i_jk; p^i_j ← p^i_jk;
    end
end

Procedure DT_Update(j, k)
begin
    D^i_jk ← RD^k_j + d_ik; p^i_jk ← rp^k_j;
    for each neighbor b do begin
        h ← j;
        while (h ≠ i or k or b) do h ← p^b_h;
        if (h = k) then begin
            D^i_jb ← D^i_kb + RD^k_j; p^i_jb ← rp^k_j;
        end
        if (h = i) then begin
            D^i_jb ← ∞; p^i_jb ← null;
        end
    end
end
```

```
Procedure Link_Change (i, k, d_ik)
when d_ik changes value do begin
    old ← d_ik;
    d_ik ← new link cost;
    for each j ∈ N do begin
        call DT_Update(j, k);
        for each j ∈ N
        do if (D^i_j > D^i_jk or k = s^i_j)
        then tag^i_j ← null;
    end
    for each j ∈ N do begin
        if (d_ik < old)
        then for each j ∈ N − i | D^i_j > D^i_jk
            do call Update(j, k)
        else for each j ∈ N − i | k = s^i_j
            do call Update(j, k)
    end
    call Send
end

Procedure RT_Update(j, DT_min)
begin
    if (D^i_{j s^i_j} = DT_min)
    then ns ← s^i_j;
    else ns ← b | {b ∈ N_i and D^i_jb = DT_min};
    x ← j;
    while (D^i_{x ns} = Min{D^i_xb ∀ b ∈ N_i}
      and (D^i_{x ns} < ∞) and (tag^i_x = null))
    do x ← p^i_{x ns};
    if (p^i_{x ns} = i or tag^i_x = correct)
    then tag^i_j ← correct;
    else tag^i_j ← error;
    if (tag^i_j = correct)
    then begin
        if (D^i_j ≠ DT_min or p^i_j ≠ p^i_{j ns})
        then add (0, j, DT_min, p^i_{j ns})
            to LIST_i(x)  ∀x ∈ N_i;
        D^i_j ← DT_min; p^i_j ← p^i_{j ns};
        s^i_j ← ns;
    end
    else begin
        if (D^i_j < ∞)
        then add (0, j, ∞, null)
            to LIST_i(x)  ∀x ∈ N_i;
        D^i_j ← ∞; p^i_j ← null;
        s^i_j ← null;
    end
end
```

Figure 4.5: HIPR Specification (cont...)

done by updating the distance and predecessor information for each destination j affected by the input event ($D^i_{jk} = D^k_j + d_{ik}$ and $p^i_{jk} = p^i_k$). In addition to this, the path to any destination j through any other neighbor which includes neighbor k is also updated. This is done by traversing the path specified by the predecessor entries reported by a neighbor from destination j towards node i. If the path implied by the predecessor reported by router b ($b \neq k$ and $b \in N_i$) to destination j includes router k, then the distance and predecessor entries are updated for that path.

The topology information within an area is transparent to nodes outside the area. Border or boundary nodes prevent the propagation of routing information outside an area by blocking such messages. All updates received by a node are processed in a similar manner.

4.3.5 Blocking Temporary Loops

A router forces its neighbors not to use it as a successor (next hop) to a given destination when it detects the possibility of a temporary loop. This is done by using an interneighbor coordination mechanism. The algorithm defines a *feasibility condition* through which a complete order of routes along a given path can be established. The *feasible distance* (FD^i_j) of router i for destination j is the smallest value achieved by the router is its own distance to j since the last time router i sent a query reporting an infinite distance to j. Between two synchronization points, the cost of the link can change or remain the same but cannot increase. This ensures that routing table loops are eliminated.

The feasible distance to a destination is initialized to infinity. At every synchronization point, the feasible distance is taken as the minimum of the existing feasible distance and the shortest path entries. Whenever the feasible distance has to be increased, an interneighbor coordination mechanism is initiated by the exchange of queries and replies.

A router is chosen as a successor to a destination only if it satisfies the following feasibility condition.

Feasibility Condition: At time t, router i can choose any router $n \in N_i(t)$ as its new successor s^i_j such that $D^i_{jn}(t) + d_{in}(t) = D_{min}(t) = \min\{D^i_{jx}(t) + d_{ix}(t) | x \in N_i(t)\}$ and $D^i_{jn}(t) < FD^i_j(t)$. If no such neighbor exists and $D^i_j(t) < \infty$, router i must keep its current successor.

4.3.6 Routing Table Updating

After procedure *DT_Update* is executed, the way in which router i updates its routing table for a given destination depends on whether router i is *passive* or *active* for that destination. A router is passive if it has a *feasible successor*, or has determined that no such successor exists and is active if it is searching for a feasible successor. A feasible successor for router i with respect to destination j is a neighbor router that satisfies FC.

When router i is passive, it reports the current value of D_j^i in all its updates and replies. While router i is active, it sends an infinite distance in its replies and queries. An active router cannot send an update regarding the destination for which it is active, because any update sent during active state would necessarily have to report an infinite distance to ensure the correct operation of the inter-neighbor synchronization mechanism used in HIPR.

If router i is passive when it processes an update for destination j, it determines whether or not it has a feasible successor, i.e., a neighbor router that satisfies FC.

If router i finds a feasible successor, it sets FD_j^i equal to the smaller of the updated value of D_j^i and the present value of FD_j^i. In addition, it updates its distance, predecessor, and successor making sure that only simple paths are used.

Router i then prepares an update message to its neighbors if its routing table entry changes. Alternatively, if router i finds no feasible successor, then it sets $FD_j^i = \infty$ and updates its distance and predecessor to reflect the information reported by its current successor. If $D_j^i(t) = \infty$, then $s_j^i(t) =$ null. Router i also sets the reply status flag $(r_{jk}^i = 1)$ for all $k \in N_i$ and sends a query to all its neighbors. Router i is then said to be *active*, and cannot change its path information until it receives all the replies to its query.

The tagging mechanism used for routing table updating ensures that only those routing table entries affected by the input event will be traversed and updated. i.e., if there is a topology change in the existing routing tree then, only nodes which are downstream to that topological change need to be updated since this change does not affect nodes upstream to the topological change. This mechanism minimizes the processing that has to be done for each update message.

A path $P_{jk}^i(t)$ is defined by the predecessors reported by neighbor k to router j stored in i's distance table at time t. To ensure loop-free paths, path traversal from j back to k is made using the predecessor information. Complete or partial path can be traversed. Path traversal ends when either a predecessor x for which $tag^i_x = correct$ or $tag^i_x = error$ or neighbor k is reached. If $tag^i_x = error$, then tag^i_j is set to error also; otherwise, the neighbor k or a correct tag must be reached in which case tag^i_j is set to correct. This mechanism ensures loop-free paths without having to traverse the entire routing table.

4.3.7 Processing of Queries and Replies

Queries and replies are processed in a manner similar to the processing of an update, as described above. If the input event that causes router i to become active is a query from its neighbor k, router i sends a reply to router k reporting an infinite distance. This happens because router k's query, by definition, reports the latest information from router k, and router i will send an update to router k when it becomes passive if its distance is smaller than infinity. A link-cost change is treated as a link failing and recovering with a new link cost.

When router i is active and receives replies from all its neighbors, it resets the reply flag ($r^i_{jk} = 0$). This means that router i's neighbors have processed the query reporting an infinite distance. Therefore, router i is free to choose any neighbor that provides the shortest distance, if there is any. If such a neighbor is found, router i updates the routing table with the minimum distance and sets $FD^i_j = D^i_j$.

A router does not wait indefinitely for replies from its neighbors because a router replies to all its queries regardless of its state. Thus, there is no possibility of deadlock due to the inter-neighbor coordination mechanism.

If router i is passive and has already set its distance to infinity ($D^i_j = \infty$), and receives an input event that implies an infinite distance to j, then router i simply updates D^i_{jk} and d_{ik} and sends a reply to router k with an infinite distance if the input event is a query from router k. This ensures that updates messages will stop when a destination becomes unreachable.

4.3.8 Example

Consider the topology with three areas, A, B and C in Figure 4.1. We concentrate on area $A1$ which contains a four-node network. Area $A1$ has three border nodes, a, b and c through which it is connected to areas $A3$ (and B), $A2$ and C, respectively. To simplify the description of how HIPR operates, a "virtual topology" of areas and routers is shown in Figure 4.6 (a). This topology consists of the destinations known to routers in area $A1$ and the links between destinations that may be known to the routers in area $A1$ depending on their routing trees. This is an eight-node topology, with each node representing either a node or an area. A link to an area indicates the existence of a link from a border node outside the area to a border node inside the area. A node indicating an area represents the border node(s) that provides inter-area connectivity to the area. Message sent by an "area node" over a link corresponds to messages sent by the corresponding border node. We explain the working of HIPR on this topology when the link connecting areas A and C fails and focus on the routing table entry for destination C.

In Figure 4.6, the number adjacent to each link represents the weight of that link; U indicates updates, Q and R represents the queries and replies respectively. The arrowhead from node x to node y indicates that node y is the successor of node x towards destination j; i.e., $s_j^x = y$. The label in the parenthesis assigned to node x indicates current distance (D_j^x) and the feasible distance from x to destination j (FD_j^x).

When link (c, C) fails, node c updates its distance table by setting the distance to C to ∞. Node c is unable to find a feasible successor which satisfies the feasibility condition to reach area C. This is because the distance to C through its other neighbors i and b is greater than the feasible distance (i.e., $D_{Ci}^c > FD_C^c$ and $D_{Cb}^c > FD_C^c$). Accordingly, node c sends a query to all its neighbors (Figure 4.6 (b)).

When nodes d and b receive the query, it updates its distance table and determines it is also not able to find a feasible successor to reach area-node C that satisfies the feasibility condition. In turn, they become active by sending out a query to their neighbors and also reply to node c with an infinite distance (Figure 4.6 (c)).

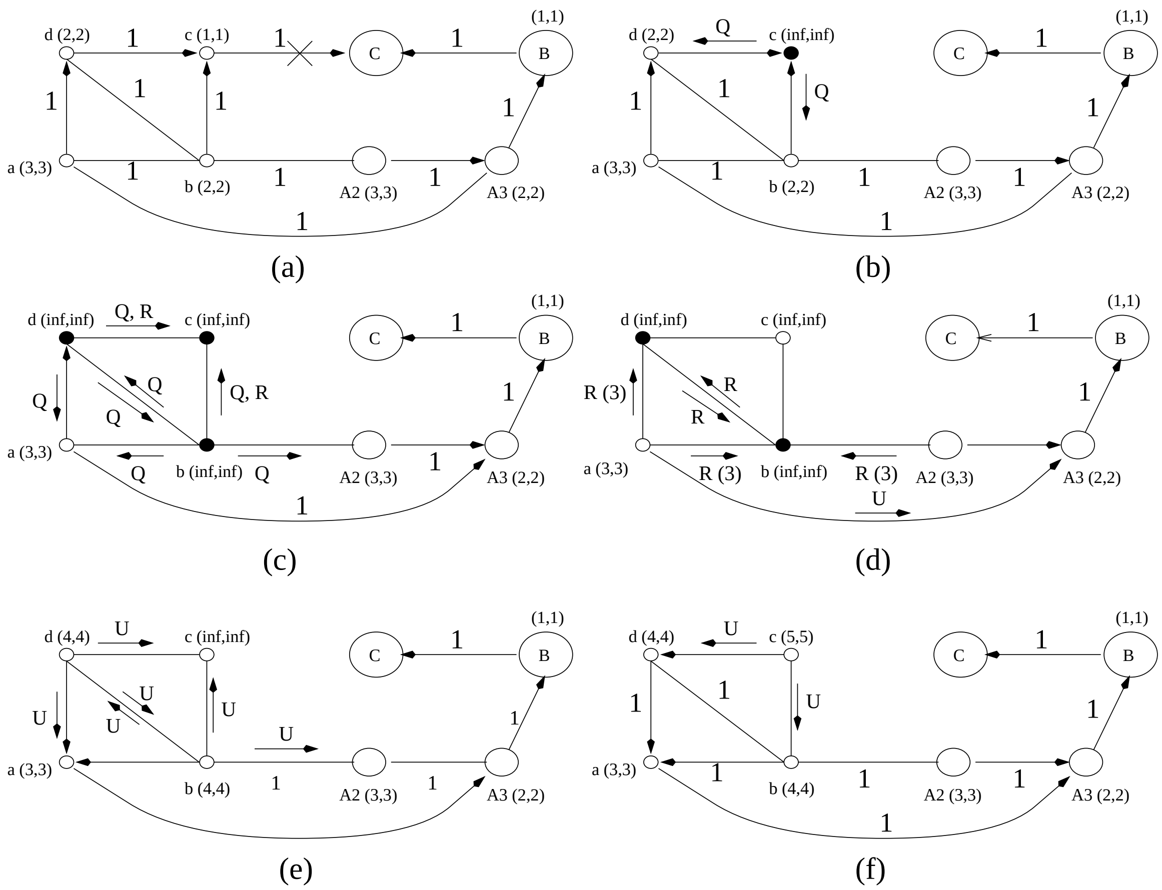

Figure 4.6: Example of HIPR

Upon receiving the queries about destination C, nodes a and $A2$ reply with a finite distance to their neighbors, because they have a feasible successor satisfying the feasibility condition. Also, node a updates its distance and routing table entries as it now uses a different path to reach C and also sends an update regarding this (Figure 4.6 (d)). On receiving replies with finite distances, nodes d and b update their distance and routing table entries and in turn send updates about their new path and distance to reach area-node C to all their neighbors, including node d (Figure 4.6 (e)). Finally, node d updates its distance and routing table entries, recomputes its feasible distance and sets node d as its successor node to reach area-node C. Updates are sent accordingly (Figure 4.6 (f)).

Any topological change within an area is transparent to the nodes outside the area unless the graph is partitioned. When a link within area $A1$ fails, HIPR will be run within area $A1$ only and this information will not be propagated to nodes outside $A1$.

4.4 Correctness of HIPR

Let $S_j(G)$ be the successor graph for each destination $j \in G$, where G defines the network topology. Loop freedom is guaranteed at all times in G if $S_j(G)$ is always a directed acyclic graph. In the steady state, when all routing tables are correct, $S_j(G)$ must be a directed tree pointed towards j. The following theorem proves that HIPR is loop free.

Theorem 3 HIPR is loop-free at every instant.

Proof: Let G be a stable topology and let the successor graph $S_j(G)$ be loop-free at every instant before time t. No loops can be created at this state unless routers change successors and modify the successor graph.

There can be two instances when a successor graph can change (a) successor graph within a area can change. (b) successor graph of the virtual network can change.

Because HIPR is the same as LPA within an area, the proof that HIPR is loop-free for case (a) follows from the proof that LPA is loop-free [GLAM96]. Because all areas in the network form a virtual heterarchical network, and each area is viewed as a single router/destination from a local node, case (b) reduces to case (a). Therefore, the successor graph is loop-free for case (b) also. This proves Theorem 3. □

To prove that HIPR converges to correct routing table entries, we assume that there is a finite time T_c after which no more resource changes (link-cost change or topology change) occur. We extrapolate the correctness properties of LPA [GLAM96] to prove that HIPR converges to correct routing tables.

Theorem 4 *HIPR is live.*

Proof: Consider the case when the network topology is stable. When a router receives a message about a topology change, it tries to find a feasible successor to the destination for which the cost metric has increased. A router sends a query to its neighbors if a feasible successor is not found. A node can be in (a) active state or (b) passive state when a query is received.

Case (a): If the node is in an active state, it immediately replies to the neighbor with an infinite distance.

Case (b): If the node is in a passive state, it tries to find a feasible successor for the destination and replies with the distance to destination.

Accordingly, when a query is received, a node either replies with a distance to destination or with an infinite distance. This implies that any router which is active by sending a query in a stable topology will become passive in a finite time (because a neighbor must answer with a reply in a finite time). This implies that HIPR is free of *deadlocks* and *live-locks* in a heterarchical network. Because the areas form a virtual heterarchical network, the same results can be extrapolated to the virtual network of areas. Therefore, HIPR is free of deadlocks and live-locks for hierarchical networks. This proves the theorem. □

Theorem 5 *After a finite time $t \geq T$, the routing tables of all routers must define the final shortest path to each destination.*

Summary of Proof: Let us assume that the result is true for a stable topology at time $T(H)$ when all messages sent by routers with shortest paths having $(H-1)$ hops $(H \geq 1)$ to a given destination j have been processed by a neighbor. Also, assume that destination j is reachable through every router.

By the inductive proof for an heterarchical network [GLAM96], the result is true for HIPR within an area.

Each router maintains information about all areas and a routing table entry for a area is treated similar to an entry of a local router/destination. This forms a virtual network where each node in the network is a area and this can be viewed as a heterarchical virtual network. The rest of the proof consists of applying a similar inductive proof as in [GLAM96] to this virtual network. □

Theorem 6 *A finite time after t, no new update messages are being transmitted or processed by routers in network G, and all entries in distance and routing tables are correct.*

Proof: On initialization, distance entries of distance and routing tables are set to infinity and predecessor entries are set to invalid. Since all nodes are disjoint when initialized, the table entries are correct.

Let the distance and routing table entries be correct at time t_1 and let the topology be stable at that time.

If the entries are not correct at time $t_2 = t_1 + t$, the only way a router can change its distance and routing table entries is by processing an update message. So, there could be three possibilities. A router might have processed an update, a query or a reply.

From Theorem 4, a finite time after an arbitrary change, router i must be passive. If router i receives an update for destination j from neighbor k at time t_i, the distance table of i has to be updated and routing table is updated if required. This update can be of two types:

(a) for a local destination

(b) for a remote destination (area)

Case (a): If the update is about a local destination, if there is no path to destination j an infinite distance is reported. If a finite distance is reported to the destination and a feasible successor is found, distance and routing tables are updated.

Case (b): If the update is about a remote destination (area), local tables are updated with the new path information.

An updated entry is added to the routing table only when the shortest path information is changed and this new information will be reported to the neighbors as routing updates.

Queries and replies are received about local destinations only (within a area). A query is always answered with a reply containing a finite distance (if feasible successor is found) or with an infinite distance (if a path does not exist). The query originator, after getting all replies from its neighbors, updates the distance and routing table entries and conveys this changed information to its neighbors.

This implies that, for any given destination, a router generates new updates or queries after it reaches its final shortest path to that destination. Because every destination must obtain its final shortest path to all destinations in finite time (from Theorem 5), the theorem is true. □

4.5 Performance of HIPR

HIPR implements Dijkstra's shortest path algorithm over a hierarchical graph in a completely distributed manner; this means that each router works directly with the hierarchical routing tree needed to carry out part of the computation of hierarchical shortest paths. In contrast, OSPF implements Dijkstra's algorithm in a replicated manner, which means that each router needs a copy of a hierarchical graph on which it runs Dijkstra's algorithm to obtain a hierarchical routing tree of its own. Therefore, HIPR can be expected to outperform OSPF. However, to obtain insight on the average performance of HIPR in a real network, we performed simulations of HIPR and OSPF, both running over the same hierarchical network topologies. Although HIPR supports arbitrary hierarchical topologies, OSPF requires the use of a backbone that interconnects areas. Accordingly, to establish a fair basis for comparison between HIPR and OSPF, we only simulated those elements of OSPF that are essential for route computation under the assumption that messages are always delivered error free over an operational link and that an infinite sequence number space can be used in OSPF to determine the validity of link state updates.

4.5.1 Network Topologies

Simulations are performed for both well-known and randomly generated backbone-based topologies. The first topology is basically a modified Doe-Esnet topology with an additional router added to the basic topology. The additional router was added to the network so that the topology can be conveniently broken up into areas, with each router placed in approximately its geographic location as it would appear on the map. The modified Doe-Esnet topology has 27 nodes and 48 links, and is shown in Figure 4.7.

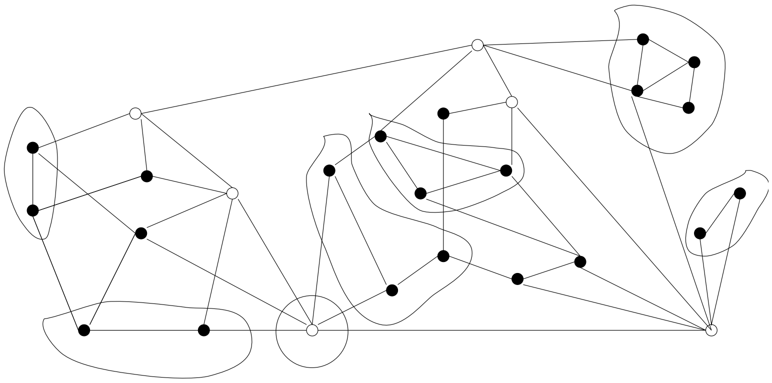

Figure 4.7: Modified Doe-Esnet Topology

To ensure that simulations of the different routing algorithms are independent of the idiosyncrasies of any specific network topology within the backbone or within areas, we also used random graphs in our study. The random graph model used to obtain random graphs is the one by Waxman [Wax88], where graphs were constructed by distributing n nodes across a cartesian coordinate grid (RG1 model). Our only interest in using Waxman's model was to obtain sparingly connected graphs to reflect the fact that, in practice, every router in an Internet is unlikely to be connected to a large percentage of the other routers.

Graphs were constructed by distributing n nodes across a cartesian coordinate grid (RG1 model). The location of each node has integer coordinates and multiple nodes were permitted to exist at any location. A Euclidean metric is used to determine the distance between each pair of nodes. The edges are introduced between pairs of nodes (u, v) with a probability that depends on the distance between them. The edge probability is given by

$$P(u,v) = \beta exp \frac{-d(u,v)}{\alpha L} \tag{4.1}$$

where $d(u, v)$ is the Euclidean distance between the nodes u and v, L is the maximum distance between two nodes, α and β are the parameters in the range (0,1] i.e., $0 < \alpha,\beta \leq 1$. Larger values of β results in graphs with higher edge densities, while small values of α increase the density of short edges relative to longer ones. The cost of each edge is equal to the distance between its endpoints. The values 0.25 and 0.2 for α and β provide graphs

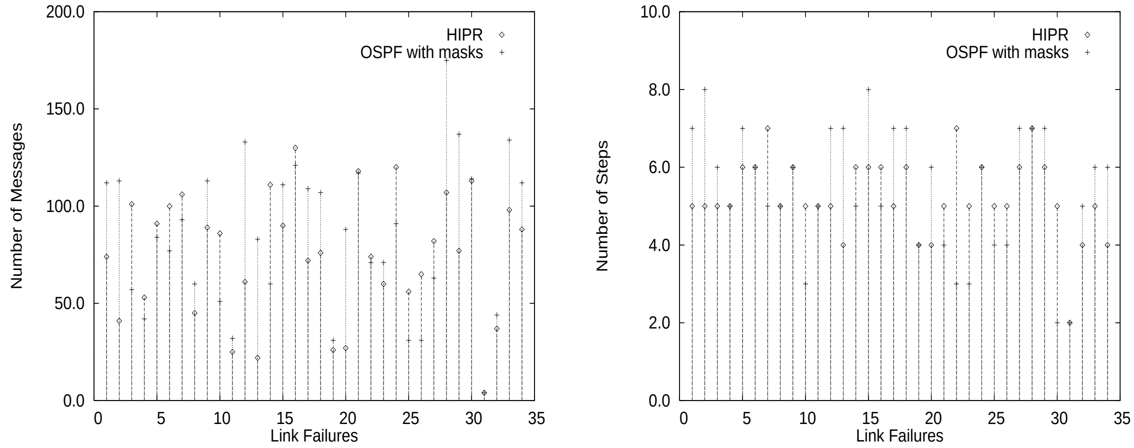

Figure 4.8: Modified Doe-Esnet Link Failure

which have the appearance roughly resembling that of geographical maps of the major nodes in the Internet.

The second topology for which we present simulation results has 10 areas and each area has 10 nodes in it. Each area has at the most 3 border nodes and a border node connects two or more areas. The interconnection among areas is based on McQuillan's approach to hierarchical routing. To generate backbone-based areas, we generated a random graph for each subnet; the backbone is also a random graph interconnecting these subnets. The random graphs are generated using Waxman's model. Our simulation network has 100 nodes and 124 links. The third topology which we have simulated is also a random graph which has approximately 300 nodes, 14 areas and the number of nodes in each area varied from 30 to 12.

4.5.2 Simulation Results

For each network, we generated test cases consisting of all single failures and recoveries for both routers and links in which the routing algorithms were allowed to converge after each change. In all cases, nodes were assumed to perform computation in zero time and links were assumed to provide one time unit of delay. The link model allows link delay and link cost to be set independently. Each unit of time therefore represents a step in which all currently available packets are processed. Although the choice of input parameters

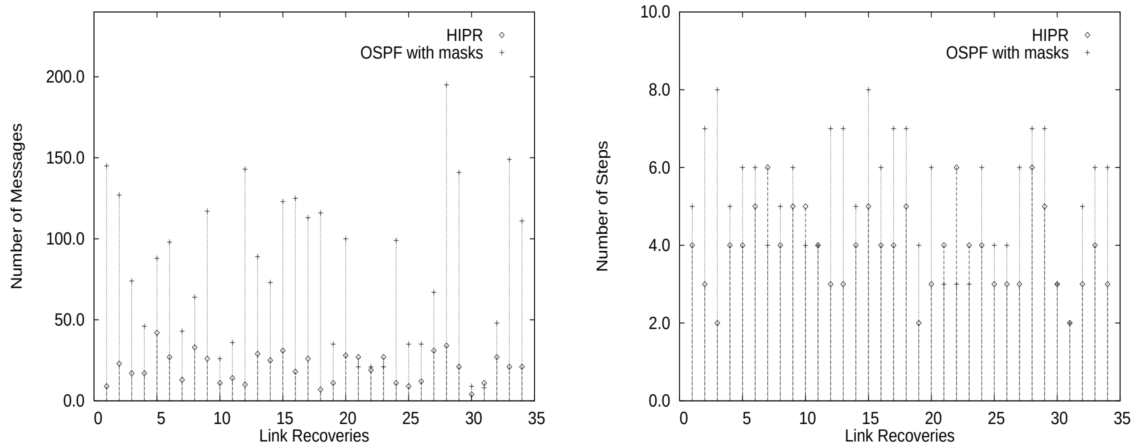

Figure 4.9: Modified Doe-Esnet Link Recovery

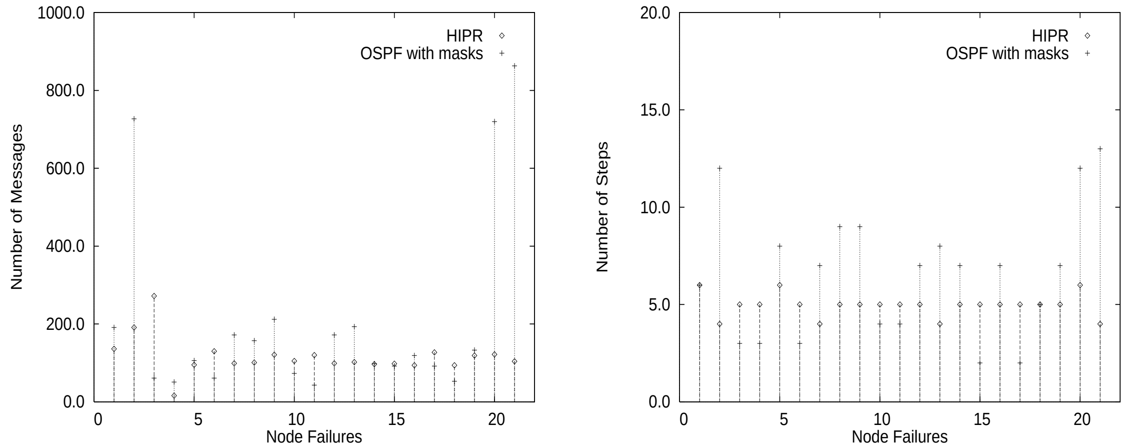

Figure 4.10: Modified Doe-Esnet Node Failure

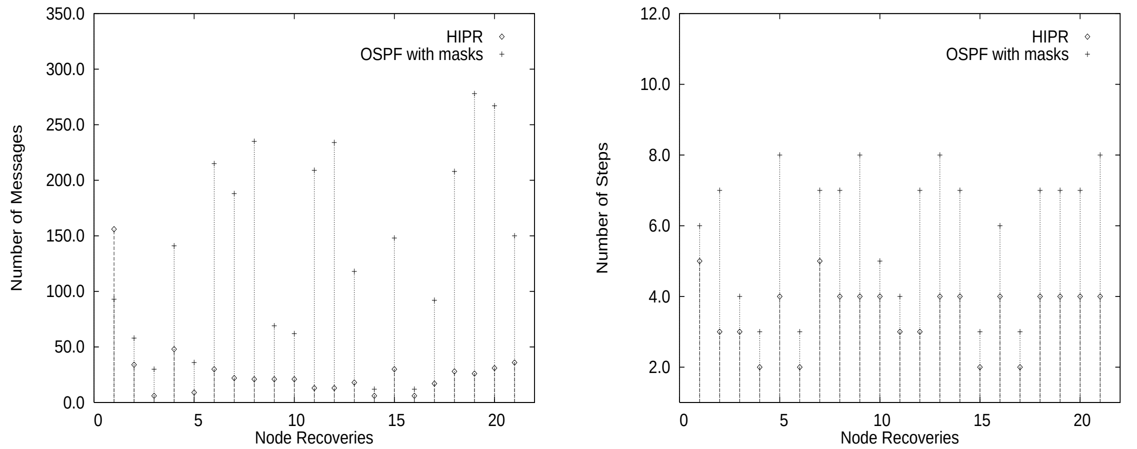

Figure 4.11: Modified Doe-Esnet Node Recovery

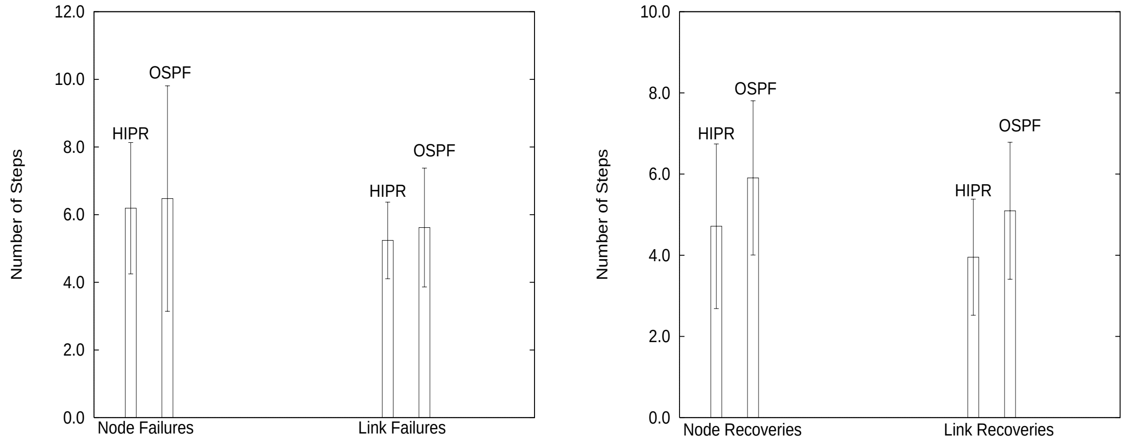

Figure 4.12: Modified Doe-Esnet: Average Duration

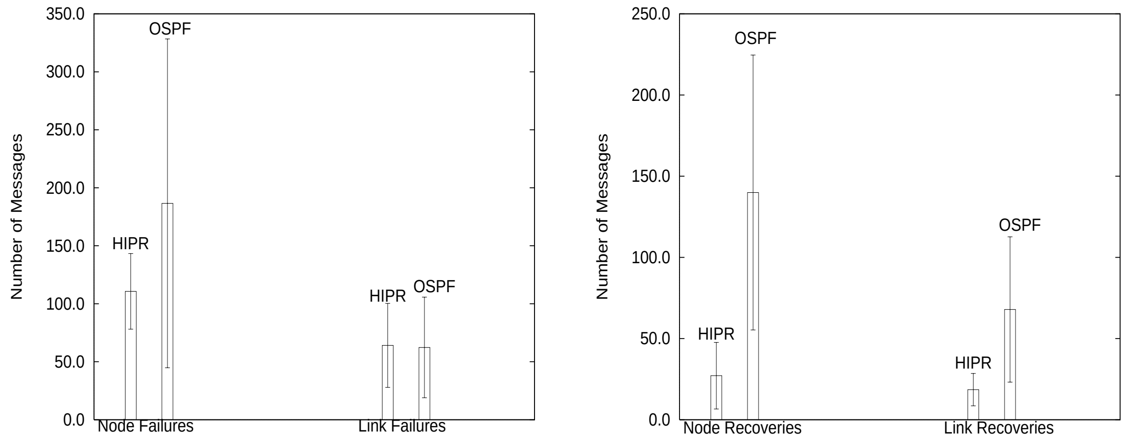

Figure 4.13: Modified Doe-Esnet: Average Number of Messages

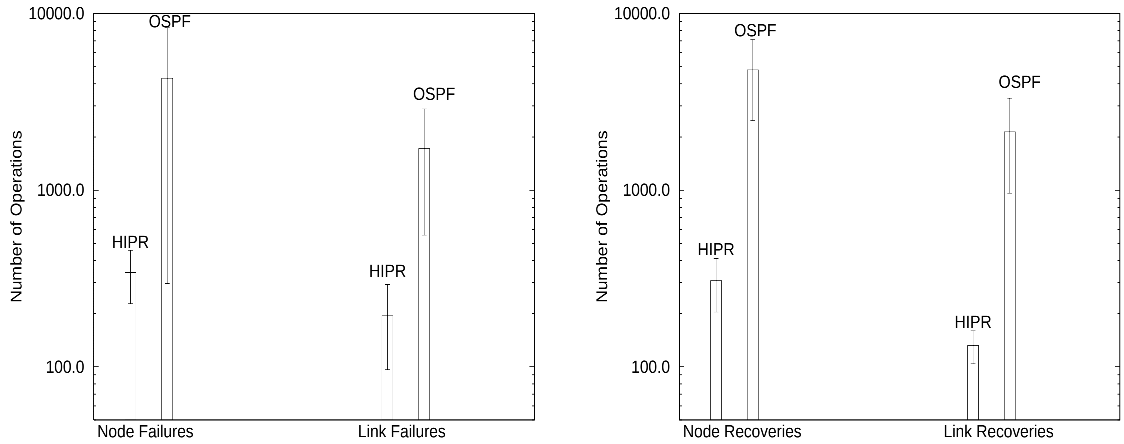

Figure 4.14: Modified Doe-Esnet: Average Number of Operations

causes the simulation to proceed synchronously, the node model treats each incoming packet

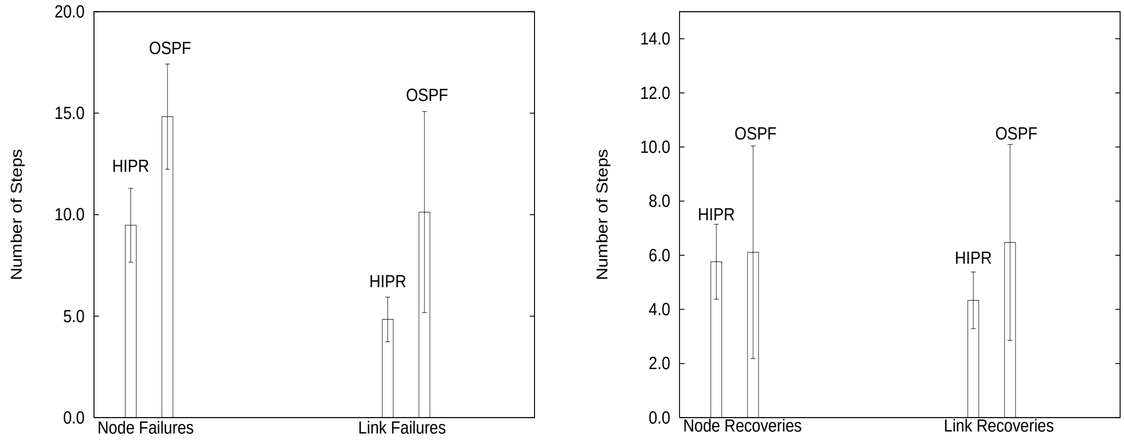

Figure 4.15: Random Graph (Topology 2): Average Duration

asynchronously. Each input event is processed independently of other events received during the same simulation step.

The average of the measured parameters is taken over all resource failures and recoveries. The algorithm was allowed to converge after each such change. Because of the distribution of values, both the mean and the standard deviation of the distribution are given. There is no sampling error for the results because all possible cases are covered.

The graphs in Figures 4.8 and 4.9 depict the number of messages exchanged and the number of steps required before each algorithm converges for every link failing and recovering in the modified Doe-Esnet topology. Similar graphs for every node failing and recovering in modified Doe-Esnet are given in Figures 4.10 and 4.11 respectively. All topology changes are performed one at a time and the algorithms were allowed to converge before the next resource change. The ordinates of the graphs represents the identifiers of the links (Figures 4.8 and 4.9) and nodes (Figures 4.10 and 4.11). The data points show the the number of messages exchanged after each resource change (graphs on the left hand side) and the number of steps needed for convergence (graphs on the right hand side) in each of these figures.

The graphs in Figures 4.12–4.14 show the average number of steps taken, average number of messages exchanged and the average number of operations performed for HIPR and OSPF before each algorithm converges in the case of modified Doe-Esnet topology. The error-bars

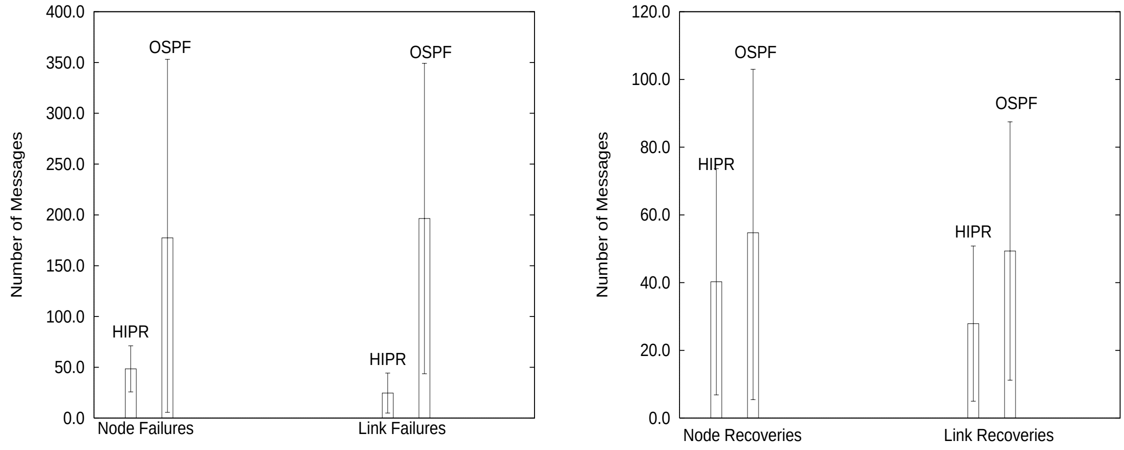

Figure 4.16: Random Graph (Topology 2): Average Number of Messages

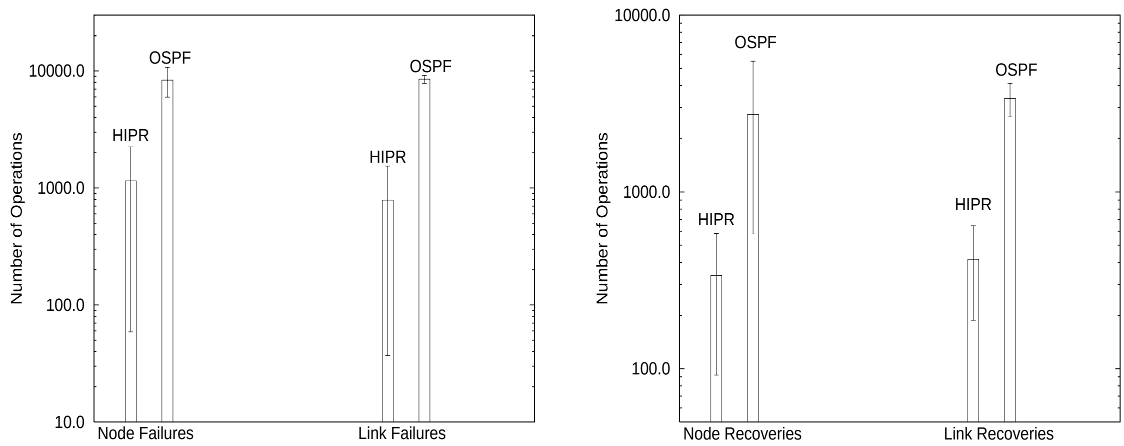

Figure 4.17: Random Graph (Topology 2): Average Number of Operations

in the graphs indicate the standard deviation of each of the mean values. Similar graphs for the random topology are shown in Figures 4.15–4.17 for Topology 2 and in Figures 4.18–4.20 for Topology 3 respectively.

HIPR outperforms OSPF in all cases considered. While the number of steps taken by HIPR to converge after a resource failure is comparable to OSPF, the number of steps taken to converge after a resource recovery is significantly better than that of OSPF. For the modified doe-esnet topology, the number of messages exchanged after node failure and recovery in case of OSPF is twice that of HIPR. The number of operations required by OSPF is an order of magnitude greater than HIPR. HIPR needs 300 operations where OSPF needs about 5000 operations. For Topologies 2 and 3 also the number of messages

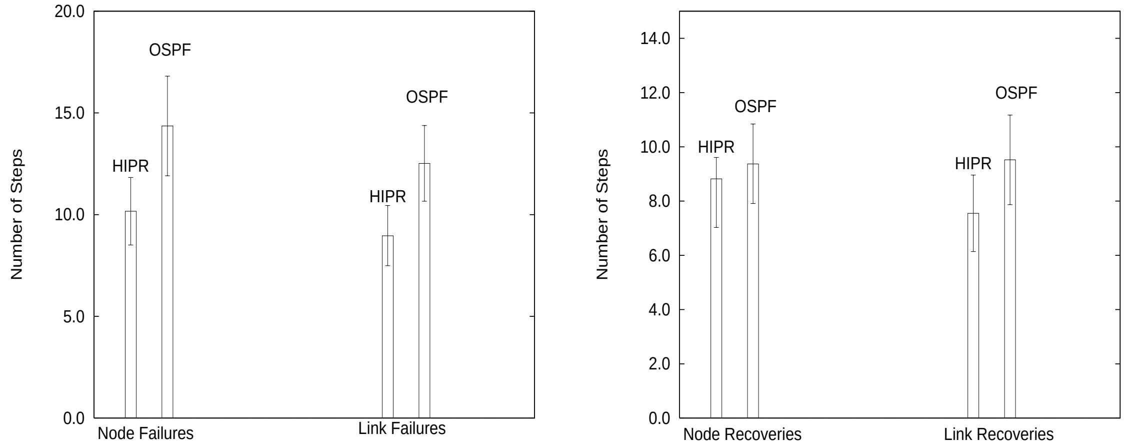

Figure 4.18: Random Graph (Topology 3): Average Duration

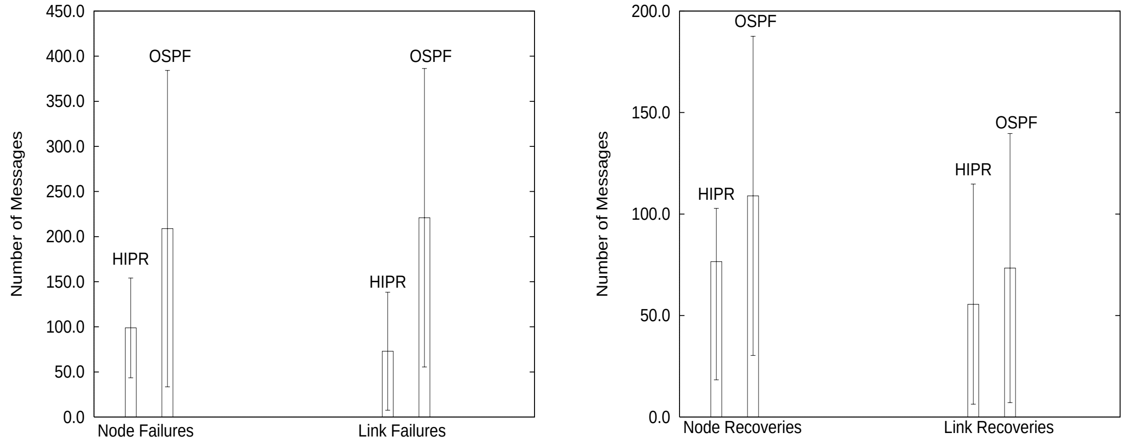

Figure 4.19: Random Graph (Topology 3): Average Number of Messages

exchanged after a resource failure in OSPF is at least twice that of HIPR and the number of messages exchanged after a resource recovery is always greater than HIPR. The number of operations in OSPF is an order of magnitude greater than that of HIPR.

This emphirical results illustrates the performance advantage provided by the distributed implementation of Dijkstra's shortest path algorithm in HIPR. It is clear that HIPR has better average performance compared to OSPF in all cases.

4.6 Summary

HIPR, the first hierarchical routing algorithm based on the maintenance and exchange of hierarchical routing trees, has been presented. The main idea of HIPR is to provide a

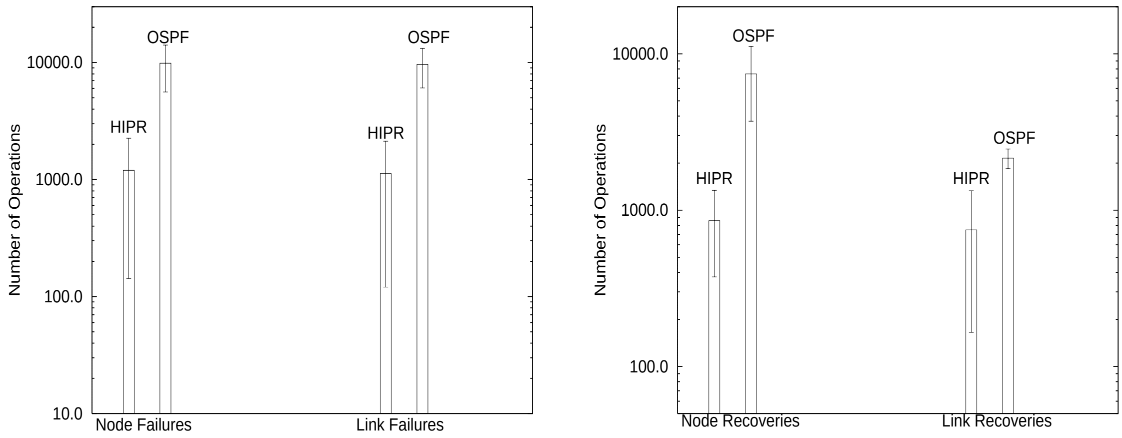

Figure 4.20: Random Graph (Topology 3): Average Number of Operations

distributed implementation of Dijkstra's shortest path algorithm over a hierarchical graph organized into areas. HIPR is an extension of LPA using McQuillan's scheme for hierarchical routing.

The performance of HIPR was compared with that of OSPF. The simulation results presented in this chapter illustrate the fact that HIPR's performance is superior to OSPF's. The number of messages exchanged in HIPR after a node failure and recovery is half that of OSPF; the number of operations required is an order of magnitude less than OSPF; and HIPR requires fewer steps to converge after a single resource change as compared to OSPF. This suggests that an Internet routing protocol based on the exchange of hierarchical routing trees would be superior to OSPF in terms of scalability and efficiency.

Chapter 5
Routing in Wireless Networks

Today's Internet technology has been extremely successful in linking huge number of computers and users. However, this technology is oriented towards computer interconnection in relatively stable operational environments, which cannot adequately support many of the emerging uses.

Wireless communication is the next step in the evolution of data networking. Wireless networks have the potential to replace the conventional wired networks where cabling proves impractical. This technology is intended to provide pervasive communication services to individuals regardless of their location. In a wireless data-network, workstations are networked using radio transmissions to accommodate mobility requirements. This requires efficient information exchange among the mobile nodes/routers. In this chapter we present an efficient routing protocol for wireless packet radio networks, which takes into account the requirements of a wireless data network. This protocol, which we call *Wireless Routing Protocol* (WRP), is based on path-finding algorithm (PFA) described in Chapter 3.

Routing protocols used in multihop packet-radio networks implemented in the past [Bea89, Bey90, LNT87] are based on shortest-path routing algorithms that have been typically based on the distributed Bellman-Ford algorithm (DBF) [BG92]. According to DBF, a routing node knows the length of the shortest path from each neighbor to every network destination and this information is used to compute the shortest path and the successor in the path to each destination. An update message contains a vector of one or more entries, each of which specifies as a minimum, the distance to a given destination. A major performance problem with DBF is that it takes a very long time to update the routing tables of network nodes after network partitions, node failures, or increase in network congestion as it has no inherent mechanism to determine when a network node should stop incrementing its distance to a given destination (counting-to-infinity problem).

As mentioned earlier in Chapter 2, counting-to-infinity problem can be overcome in three ways in the existing Internet routing protocols. However, there are significant differences between wireless networks and wired internets in which Internet routing protocols are used. A wired Internet has relatively high bandwidth and topology that changes infrequently; in contrast, wireless networks have mobile nodes and have limited bandwidth for network control. Accordingly, flooding, multihop internodal synchronization and the specification of complete path information would incur too much overhead in a multihop packet radio network with a dynamic topology. On the other hand, the routing protocols based on DBF or modifications of DBF would take a long time to converge and the frequent topology changes in a wireless network with mobile nodes make the looping problem of DBF unacceptable. Therefore, there is a need for a new routing protocol, which is devoid of all these drawbacks.

In the recent past, a number of efforts have been made to address the limitation of DBF and topology broadcast in mobile wireless networks. One such effort is the DSDV protocol [PB94]. In this protocol, each mobile host, which is a specialized router, periodically advertises its view of the interconnection topology to other mobile hosts within the network to maintain an up to date network state information. Unfortunately, in DSDV a node has to wait until it receives the next update message originating from the destination in order to update its distance-table entry for that destination. This implicit destination-centered synchronization suffers from the same latency problems as DUAL (and similar algorithms based on explicit synchronization). DSDV uses both periodic and triggered updates for updating routing information; also, DSDV floods the sequence numbers originated by each destination, which could cause excessive communication overhead.

A distributed routing algorithm for mobile wireless networks based on diffusing computations has been proposed by Corson and Ephremides [CE95]. This protocol relies on the exchange of short control packets forming a *query-reply* process. It also has the ability to maintain multiple paths to a given destination. This is a destination-oriented protocol in which separate versions of the algorithm run independently for each destination. Routing is source-initiated, which means that routes are maintained by those sources that actually desire routes. Even though this algorithm provides multiple paths to a destination, because of the query-based synchronization approach to achieve loop-free paths, the communication complexity could be high.

Path-finding algorithms are an attractive approach for wireless networks, because they eliminate counting-to-infinity problem. As seen in Chapter 3, they converge faster than the state of the art routing algorithms. In the remaining of this chapter we describe a wireless routing protocol (WRP) based on PFA illustrating the key aspects of the protocol's operation and present some performance results.

5.1 Wireless Routing Protocol

5.1.1 Overview

WRP is designed to run on top of the medium-access control protocol of a wireless network. Update messages may be lost or corrupted due to changes in radio connectivity or jamming. Reliable transmission of update messages is implemented by means of retransmissions. After receiving an update message free of errors, a node is required to send a positive acknowledgment (ACK) indicating that it has a good radio connectivity and has processed the update message. Because of the broadcast nature of the radio channel, a node can send a single update message to inform all its neighbors about changes in its routing table; however, each such neighbor sends an ACK to only the originator node.

In addition to ACKs, the connectivity can also be ascertained with the receipt of any message from a neighbor (which need not be an update message). To ensure that connectivity with a neighbor still exists when there are no recent transmissions of routing table updates or ACKs, periodic update messages without any routing table changes (null update messages) are sent to the neighbors. These messages are also known as *Hello* messages. The time interval between two such null update messages is the *HelloInterval*.

If a node does not receive any message from a neighbor for a specified amount of time (e.g., three or four times the HelloInterval known as the *RouterDeadInterval*), the node will assume that connectivity with that neighbor is lost.

5.1.2 Information Maintained at Each Node

For the purpose of routing, each node maintains a *distance table*, a *routing table*, a *link-cost table* and a *message retransmission list*.

The distance table of node i is a matrix containing, for each destination j and each neighbor of i (say k), the distance to j (D^i_{jk}) and the predecessor (p^i_{jk}) reported by k.

The routing table of a node i is a vector with an entry for each known destination j which specifies:

- The destination identifier (j)
- The distance to the destination (D^i_j)
- The predecessor of the chosen shortest path to j (p^i_j)
- The successor of the chosen shortest path to j (s^i_j)
- A marker (tag^i_j) used to update the routing table entries; it specifies whether the entry corresponds to a simple path (tag^i_j = correct), a loop (tag^i_j = error) or a destination that has not been marked (tag^i_j = null).

The link-cost table of node i lists the cost of relaying information through each neighbor k, and the number of periodic update periods that have elapsed since node i received any error-free messages from k.

The cost of a failed link is considered to be infinity. The way in which costs are assigned to links is beyond the scope of this specification. As an example, the cost of a link could simply be 1 reflecting the hop count, or the addition of the latency over the link plus some constant bias. The cost of the link from i to k (i,k) is denoted by l^i_k.

Procedure Init1
when router i initializes itself
do begin
  set a link state table with costs of adjacent links;
  $N \leftarrow i$; $N_i \leftarrow x \mid l_x^i < \infty$;
  for each $(x \in N_i)$
  do begin
    $N_i \leftarrow N \cup x$; $tag_x^i \leftarrow null$;
    $s_x^i \leftarrow null$; $p_x^i \leftarrow null$; $D_x^i \leftarrow \infty$
  end
  $D_i^i \leftarrow 0$; $s_i^i \leftarrow null$; $p_i^i \leftarrow null$; $tag_i^i \leftarrow correct$
  for each $j \in N$ call **Init2**(x, j)
  for each $(n \in N_i)$ **do** add $(0, i, 0, i)$ to $LIST_i(n)$
  $x \leftarrow$ retransmission time; $y \leftarrow$ hello count;
  $z \leftarrow$ retransmission count;
  call **Send**
end

Procedure Init2(x, j)
begin
  $D_{jx}^i \leftarrow \infty$; $p_{jx}^i \leftarrow null$; $s_{jx}^i \leftarrow null$; $seqno_{jx}^i \leftarrow 0$
end

Procedure Send
begin
  for each $(n \in N_i)$
  do begin
    if ($LIST_i(n)$ is not empty)
    then send messages with $LIST_i(n)$ to n
    empty $LIST_i(n)$
  end
end

Procedure Message
when router i receives a message on link (i, k)
begin
  if $(k \notin N_i)$ **do**
  begin
    $N_i \leftarrow N_i \cup k$;
    $l_k^i \leftarrow$ cost of new link;
    if $(k \notin N)$ **begin**
      $N \leftarrow N \cup k$; $tag_k^i \leftarrow null$;
      $D_k^i \leftarrow \infty$; $p_k^i \leftarrow null$; $s_k^i \leftarrow null$;
      for each $x \in N_i$ **do** call **Init2**(x, k)
    end
    for each (i, k, l_k^i) **do**
      send update$(0, k, D_k^i, p_k^i)$
  end
  reset HelloTimer;
  for each entry $(u_j^k, j, RD_j^k, rp_j^k) \mid i \neq j$
  do begin
    if $(j \notin N)$
    then begin
      if $(RD_j^k = \infty)$ **then** delete entry
      else begin
        $N \leftarrow N \cup j$;
        for each entry $x \in N_i$ call **Init2**(x, j)
        $tag_j^i \leftarrow null$; call **DT**
      end
    end
    else begin
      $tag_j^i \leftarrow null$;
    end
  end
  for each entry $(u_j^k, j, RD_j^k, rp_j^k)$ left $\mid i \neq j$
  do case of u_j^k
    0: call **Update**(j, k)
    1: call **ACK**(j, k)
  end
  call **Send**
end

Procedure Create_RList$(seqno)$
begin
  $seqno \leftarrow seqno + 1$; $NeighborSet \leftarrow N_i$
  $bitmap[] \leftarrow 0$; $RetransmissionTimer \leftarrow x$
  add updates to RList
end

Procedure Delete_RList$(seqno)$
begin
  set $bitmap[seqno] \leftarrow 1$; $delete \leftarrow 1$
  for all $n \in N_i$ **begin**
    if $(bitmap[seqno] = 0)$ $delete \leftarrow 0$;
  end
  if $(delete = 1)$ delete RList[seqno] **end**

Procedure Update_RList$(seqno)$
begin
  reset RetransmissionTimer
  send update $RList[seqno]$;
end

Procedure Clean_RList $(seqno)$
begin
  for all entries in $RList$
    delete $RList[seqno]$;
end

Procedure Connectivity
when HelloTimer expires
begin
  $HelloCount[k] \leftarrow HelloCount[k] + 1$;
  if $(HelloCount[k] < y)$ **then**
    reset HelloTimer;
  else begin
    $N_i \leftarrow N_i - k$
    call **Delete_RList**(k)
    $l_k^i \leftarrow \infty$
    $tag_k^i \leftarrow null$
    delete column for k in distance table
    update routing table
  end
end

Procedure TimeOut(i, k)
when RetransmissionTimer expires
begin
  RetransmissionCounter $\leftarrow$ RetransmissionCounter - 1;
  if (RetransmissionCounter $<$ z)
    call **Update_RList**(k)
  else begin
    $N_i \leftarrow N_i - k$
    call **Delete_RList**(k)
    $l_k^i \leftarrow \infty$
    $tag_k^i \leftarrow null$
    delete column for k in distance table
    update routing table
  end
end

Procedure DT
when distance table update has to be done
begin
  $D_{jk}^i \leftarrow l_k^i + D_j^k$; $p_{jk}^i \leftarrow p_j^k$;
(2) **for all** neighbors b
  do begin
    if k is in the path from i to j in
      the distance table through neighbor b
    then $D_{jb}^i \leftarrow D_{kb}^i + D_j^k$; $p_{jb}^i \leftarrow p_j^k$
  end
end

Figure 5.1: Protocol Specification

Procedure ACK(n)
when router i receives an ACK on link (i, k)
begin
call **Delete_RList**(n);
RetransmissionCounter $\leftarrow$ z;
end

Procedure Update(i, k)
when router i receives an update on link (i, k)
begin
send ACK to neighbor k
RetransmissionCounter $\leftarrow$ z;
RetransmissionTimer $\leftarrow$ x;
(0) **begin**
update=0;
$RTEMP^i \leftarrow \phi$;
$DTEMP^{i,b} \leftarrow \phi$ for all neighbors b
(1) **for each** triplet (j, D_j^k, p_j^k) in $V^{k,i}, j \neq i$ **do**
call procedure **DT**
(3) **begin**
if there are b and j such that
$(D_{jb}^i < D_j^i)$ or $((D_{jb}^i > D_j^i)$ and $(b = s_j^i))$
then call **RT_Update**
end
(4) **begin if** $(RTEMP^i \neq \phi)$ **then**
for each neighbor b **do begin**
for each triplet $t = (j, D_j^i, p_j^i)$ in $RTEMP^i$
do begin
if b is not in the path from i to j
then $DTEMP^{i,b} \leftarrow DTEMP^{i,b} \cup t$;
end
send $DTEMP^{i,b}$ to neighbor b;
end
end
end

Procedure RT_Update
when routing table has to be updated
begin
find minimum of the distance entries DT_{min}
if $(D^i_{j\, s^i_j} = DT_{min})$ **then** $ns \leftarrow s_j^i$
else $ns \leftarrow b \mid \{b \in N_i$ and $D_{jb}^i = DT_{min}\}$;
$x \leftarrow j$;
while $(D^i_{x\ ns} = Min\{D^i_{xb}\ \ \forall\ \ b \in N_i\}$
and $D^i_{xns} < \infty$ and $tag^i_x = null)$
do $x \leftarrow p^i_{x\ ns}$;
if $(p^i_{x\ ns} = i$ or $tag^i_x =$ correct)
then $tag^i_j \leftarrow$ correct **else** $tag^i_j \leftarrow$ error
if $(tag^i_j =$ correct) **then begin**
if $(D^i_j \neq DT_{min}$ or $p^i_j \neq p^i_{j\ ns})$ **then begin**
$seqno \leftarrow seqno + 1$;
add $(0,\ j,\ DT_{min},\ p^i_{j\ ns},\ seqno)$ to $LIST_i(x)\ \ \forall x \in N_i$;
call **Clean_RList**($seqno$)
call **Create_RList**($seqno$)
end
$D^i_j \leftarrow DT_{min}$; $p^i_j \leftarrow p^i_{j\ ns}$; $s^i_j \leftarrow ns$
end
else begin
if$(D^i_j < \infty)$ **then begin**
$seqno \leftarrow seqno + 1$;
add $(0,\ j,\ \infty,$ null, $seqno)$ to $LIST_i(x)\ \ \forall x \in N_i$;
call **Clean_RList**($seqno$)
call **Create_RList**($seqno$)
end
$D^i_j \leftarrow \infty$; $p^i_j \leftarrow$ null; $s^i_j \leftarrow$ null
end
end

Figure 5.2: Protocol Specification (Cont..)

The message retransmission list (MRL) specifies one or more retransmission entries, where the m^{th} entry consists of the following:

- The sequence number of an update message
- A retransmission counter that is decremented every time node i retransmits an update message
- A flag indicating whether node i has received an ACK from neighbor k to the update message sent to k (present in the retransmission entry)
- The list of updates sent in the update message

The above information permits node i to know which updates of an update message (each update message may contain a list of updates) have to be retransmitted when the timer expires and which neighbors should be requested to acknowledge such retransmission. Node i retransmits the list of updates in an update message when the retransmission timer of the corresponding entry in the MRL expires. The retransmission counter of a new entry in the MRL is set equal to a small number (e.g., 3 or 4).

5.1.3 Information Exchanged among Nodes

In WRP, nodes exchange routing-table update messages (which we call "update messages" for brevity) that propagate only from a node to its neighbors. An update message contains the following information:

- The identifier of the sending node.
- A sequence number assigned by the sending node.
- An *update list* of zero or more updates or ACKs to update messages. An update entry specifies a destination, a distance to the destination, and a predecessor to the destination. An ACK entry specifies the source and sequence number of the update message being acknowledged.
- A *response list* of zero or more nodes that should send an ACK to the update message.

In the event that the message space is not large enough to contain all the updates and ACKs that a node wants to report, they are sent in multiple update messages. An example of this event can be the case in which a node identifies a new neighbor and sends its entire routing table.

The response list of the update message is used to avoid the situation in which a neighbor is asked to send multiple ACKs to the same update message, simply because some other neighbor of the node sending the update did not acknowledge.

The first transmission of an update message must ask all neighbors to send an ACK, of course, and this is accomplished by specifying the "all-neighbors address," which consists of all 1's.

When the update message reports no updates, the "empty address" is specified; this address consists of all 0's and instructs the receiving nodes not to send an ACK in return. This type of update message is used as a "hello message" from a node to allow its neighbors to know that they maintain connectivity, even if no user messages or routing-table updates are exchanged.

As we explain subsequently, an ACK entry refers to an entire update message, not an update entry in an update message, in order to conserve bandwidth.

5.1.4 Routing-Table Updating

Figures 5.1 and 5.2 specify important procedures of WRP used to update the routing and distance tables.

A node can decide to update its routing table after either receiving an update message from a neighbor, or detecting a change in the status of a link to a neighbor. When a node i receives an update message from its neighbor k, it processes each update and ACK entry of the update message in order.

In WRP, a node checks the consistency of predecessor information reported by *all* its neighbors each time it processes an event involving a neighbor k. i.e., the distance table is updated for all entries referring directly to neighbor k *and* all other entries that refer to node k indirectly (k is in the path of a destination). In contrast, all previous path-finding algorithms [CRKGLA89, Hum91, RF91] check the consistency of the predecessor only for the neighbor associated with the input event. This unique feature of WRP accounts for its fast convergence after a single resource failure or recovery as it eliminates more temporary looping situations than previous path-finding algorithms.

Processing an Update: To process an update from neighbor k regarding destination j, the distance and the predecessor entries in the distance table are updated. A flag (tag) is set to specify that this entry in the table has been changed. A unique feature of WRP is that node i also determines if the path to destination j through any of its other neighbors $\{b \in N_i | b \neq k\}$ includes node k. If the path implied by the predecessor information reported by node b includes node k, then the distance entry of that path is also updated as

$D^i_{jb} = D^i_{kb} + D^k_j$ and the predecessor is updated as $p^i_{jb} = p^k_j$. Thus, a node can determine whether or not an update received from k affects its other distance and routing table entries.

To update its distance and predecessor for destination j (procedure RT_Update), node i chooses a neighbor p that has reported routing information such that:

- The path from p to j (which is implied by the predecessor information reported by p) does not include node i
- $D^i_{jp} \le D^i_{jx}$ for any other neighbor x, and $D^i_{yp} \le D^i_{yx}$ for any other neighbor x and for every node y in the path from i to j.

The above means that node i chooses node p as its successor to a destination j if that neighbor appears to offer a smallest-cost loop-free path to j on all the intermediate nodes in the path to j.

When node i sends an update message, it updates its message retransmission list. For each destination j for whom an update is being reported, node i sets the ack-required flag for all its neighbors. It also adds an entry in the message-retransmission list containing the sequence number given to the update message, and starts the retransmission timer for that entry.

Sending New and Retransmitted Update Messages: Node i sends a new update message after processing updates from its neighbors or detecting a change in the status of the link to a neighbor. Whenever node i sends a new update message, it must

- Add an entry in the MRL for the new update message
- Delete the updates in the existing entries of MRL for the updates that are included in the new update message
- Initialize the retransmission counter of ther new update messages entry in the MRL to a maximum value.

When the list of updates of a MRL entry is outdated by the transmission of a new update message, node i erases the old entry from the MRL.

When the retransmission timer for a entry m in the MRL expires, node i retransmits the update message as an update list containing the list of updates of the retransmission entry, and a response list specifying those neighbors who did not acknowledge the update message earlier (i.e., every neighbor k for whom $a^i_{km} = 1$). The retransmission counter of that entry in the MRL is decremented.

Note that, if we do not use the retransmission counter, based on the above retransmission strategy, there is no limit on the number of times node i would retransmit an update message to an existing neighbor. However, as we discuss below, node i stops considering node k as its neighbor after it fails to communicate with it for some finite amount of time.

Retransmission counter limits the number of times an update can be retransmitted to a neighbor. Each time an update is retransmitted, the corresponding retransmission counter is decremented. If an ACK is not received from a neighbor before the retransmission counter becomes zero, then that neighbor is assumed to be no longer existent.

Processing an ACK: An ACK entry in an update message refers to another update message, i.e., it acknowledges all the updates included in the update message bearing the referenced sequence number. Therefore, it is up to the node whose update message is being acknowledged to ascertain which updates are implied by a received ACK.

To process an ACK from neighbor k, node i scans its MRL for the sequence number matching the sequence number specified in the ACK received. When a match is found, node i resets the ack-required flag for neighbor k; if $a^i_{pm} = 0$ for entry m and every neighbor p of node i; the retransmission entry is deleted if all entries of the retransmission entry has been acknowledged. This scheme obtains short ACKs at the expense of additional processing.

Node i may receive an ACK for an update message whose retransmission entry has been erased after sending a more recent update message for the same destinations. In that case, node i simply ignores the ACK.

Handling Topology and Link-Cost Changes: To ensure that nodes know that they have connectivity even when they do not transmit user messages or routing-table updates for some time, every node must periodically send an update message reporting no changes (*hello messages*). Acknowledgments are not required for such update messages, and they can be very short (e.g., a byte for control information and a byte for the node identifier, since the control information can imply that there is no sequence number, update list, or response list in the message). Alternatively, a node may retransmit an update message if it is not too long. On initialization, a node transmits a hello message.

Given that short periodic update messages are transmitted by every node, the failure of a link to a neighbor is detected by the lack of any user or update messages being received from that neighbor over a period of time equal to a few update-message transmission periods. Similarly, new links and nodes are detected by means of update messages or user messages.

When node i receives an update or user message from node k and node k is not listed in its routing table or distance table, it adds the corresponding entry to its distance or routing table for destination k. An infinite distance to all destinations through node k is assumed, with the exception of node k itself and those destinations reported in node k's updates, if the message received from k was an update message. In addition, node i notifies node k of the information in its routing table. This information can be transmitted in one or multiple update messages that only node k needs to acknowledge.

When a link fails or a link-cost changes, node i recomputes the distances and predecessors to all affected destinations, and sends to all its neighbors an update message for all destinations whose distance or predecessor have changed.

5.2 Correctness of WRP

In this section, we show that the basic routing algorithm used in WRP is correct. The following assumptions are made on the behavior of links and routers for the working of WRP.

1. Messages are transmitted reliably. A lower-level protocol is responsible for maintaining the status of the link.

2. Messages are sent by a router over a link only when the link is perceived as being up.

3. A router that is not functional cannot receive or send any messages.

4. All routers are initially down.

5. Update messages received by a router are processed in the order of their arrival (FIFO).

6. Link lengths are always positive and a failed link has an infinite length.

7. Time T is defined such that between the time interval 0 and T links and routers go up and down and the cost of the link changes; at time T, links have the same status at both ends and there are no changes after time T.

For simplicity, the proof assumes that all update messages sent over an operational link are received correctly. In practice, WRP handles errors by means of retransmissions. In terms of the correctness proof, the effect of retransmissions is that of added delay in the delivery of an update message to a neighbor, and a link fails when a given number of retransmissions have been attempted. In essence, this proof is very much similar to that of the path-finding algorithm (PFA) on which WRP is based is correct. The proof of correctness is given in [MGLA96].

WRP's time complexity is $O(h)$ in the worst-case, where h is the height of the routing tree. *Time complexity* is defined as the largest time that can elapse between the moment T when the last topology change occurs and the moment at which all the routers have final shortest path to all other routers. *Communication complexity* is defined as the maximum number of node identities exchanged (messages) after time T before the final graph is reached.

5.3 Simulation Results

We have studied the average-case performance of WRP in a dynamic environment using *Drama*. The performance of the basic routing algorithm used in WRP has been compared with that of DUAL and ILS. In order to simulate mobility, the connectivity of a mobile node is said to be lost when a node does not hear from a mobile node for a certain period of time. The connectivity with a node will be reestablished when a node hears from a mobile node again. Mobility is modeled as an arbitrary set of failures and recoveries of a mobile node at random points in time. All simulations are done assuming unit propagation time and zero packet processing time at each node. If a mobile node fails when the packets are in transit, the packets are assumed to get dropped.

Our goal is to compare the performance of WRP against the performance of routing protocols based on DBF, DUAL, and ILS. To reduce the complexity of the simulation, we have eliminated those features of the protocols that were common to all; these features concern the reliable transmission of updates over unreliable links, and the identification of neighbors. Accordingly, our simulation assumed that, for any of the protocols simulated, any update message sent over an operational link is received correctly, and that a node always receives enough user messages to know that it continues to have connectivity with a neighbor. According to these assumptions, there is no need to account for acknowledgments, retransmissions of updates, or periodic transmissions of update messages.

However, our intent in running the simulations was to obtain insight on the comparative overhead of different protocols that necessarily require the transmission of acknowledgments to update messages. We approached this problem in the following manner: In a wireless packet radio network, the same update messages sent by a node is received by all its neighbors, i.e., each update message is broadcast to a node's neighbors. However, to guarantee the reliable transmission of updates, each neighbor must send an acknowledgment to the sender of the update. Therefore, under the assumption that no errors or collisions occur in the network channel, counting the number of acknowledgments received for a single update broadcast to all neighbors is much the same as counting the number of updates sent by a node to its neighbors on a point-to-point basis and with no acknowledgments—the two counts differ only by one. Accordingly, we simulated the routing protocols' operation in a wireless network using the same point-to-point links typical of wireline networks. The message count obtained from the simulation runs is not the exact number of updates and acknowledgments sent by each protocol, but accurately reflects the relative differences among protocols.

The resulting simplified version of WRP we simulated is simply the path finding algorithm (PFA), and is the same basic algorithm first described in [MGLA94]. Similarly, ILS, DBF, and DUAL correspond to the ideal case of the best protocols that could be designed based on these algorithms.

The simulations were run on several network topologies such as *Los-Nettos*, *Nsfnet* and *Arpanet*. We chose these topologies to compare the performance of routing algorithms for well-known cases given that we cannot sample a large enough number of networks to make statistically justifiable statements about how an algorithm scales with network parameters. The Los-Nettos topology has 11 nodes, a diameter of 4 hops, and each node has at most four neighbors. The Nsfnet topology has 13 nodes, a diameter of 4 hops, and each node has at most 4 neighbors. The Arpanet topology has 57 nodes, a diameter of 8 hops, and each node has a maximum of four neighbors.

For the routing algorithms under consideration, there is only one shortest path between a source and a destination pair and we do not consider null paths from a node to itself. Data are collected for a large number of topology changes to determine statistical distribution. The statistics has been collected after each failure and recovery of a link. To obtain the average figures, we make each link (or node) in the network fail and count the number of steps and messages required for each algorithm to converge. Then the same link (node) is made to recover and the process is repeated. The average is taken over all failures and recoveries. Again, this message count is not exact, but the relative difference from one protocol to another is accurate.

5.3.1 Dynamics with Mobile Nodes

We modeled mobility in the simulation by making the links fail and come back up arbitrarily at random times. The network is assumed to be fully connected with potential links. At startup, the topology is initialized to some well known topology, such as *Los-Nettos*, *Nsfnet* or *Arpanet*. After initialization, to simulate the movement of a node, a node is assumed to have failed at its previous location and reappear in its new location. Node failure is simulated as all the links associated with that node going down at the same time. The gradual movement of a node from one location to another is simulated by means of link failures and additions. When a link fails, it can be assumed that a node is no longer in the neighborhood of its previous neighbor. The addition of a new link is viewed as a movement of a node wherein, a node reappears in the new neighborhood.

The links are chosen at random from the set of all the existing links in the fully connected network. Selecting any particular link is equally likely. The probability of a link failing or recovering is also equally likely. We also have imposed an additional condition in our simulations that a node at any given time cannot have more than x neighbors. Here, x indicates the degree of the node. This condition is imposed in order to make sure that all the links pertaining to one node alone will not be active. This helps in simulating the mobility more closely. This, of course, is only an approximation of the more gradual topology changes that would be experienced in a real mobile network.

The average number of messages and the average message length for each of these algorithms are obtained by varying the interarrival time between two events (Figures 5.3–5.5). An event can be either a link failure or a link recovery. For the purpose of event generation, we consider a fully connected topology and start off with a given initial topology. Since any random link can fail or recover at any time, our model simulates mobility closely.

The above results indicate that the routing algorithm of WRP outperforms all other algorithms which we have simulated, namely, DBF, DUAL and ILS. We were not able to simulate ILS for the Arpanet topology due to limited resources. The statistics about the average number of messages and the average message length have been collected for all the above mentioned topologies for all the four algorithms by varying the interarrival time between events (failures and recoveries).

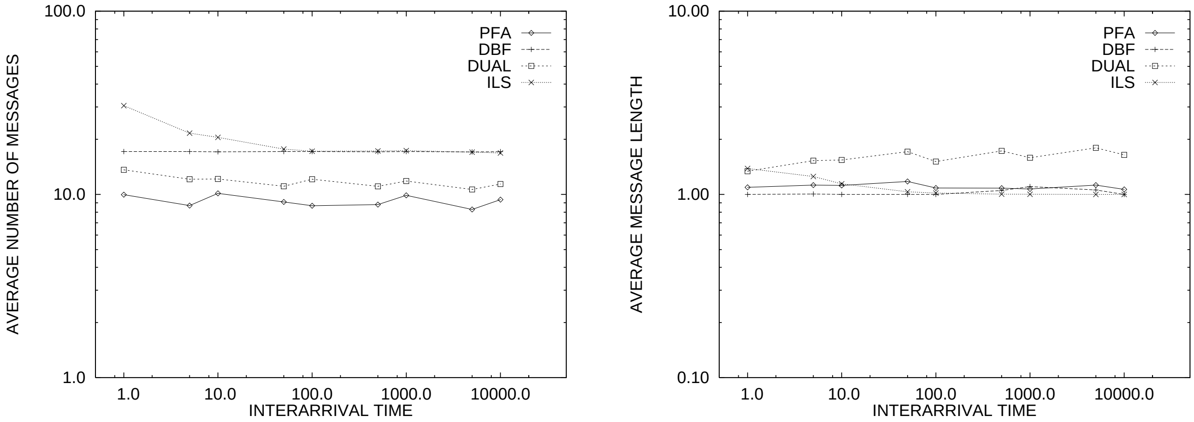

Figure 5.3: Los-Nettos

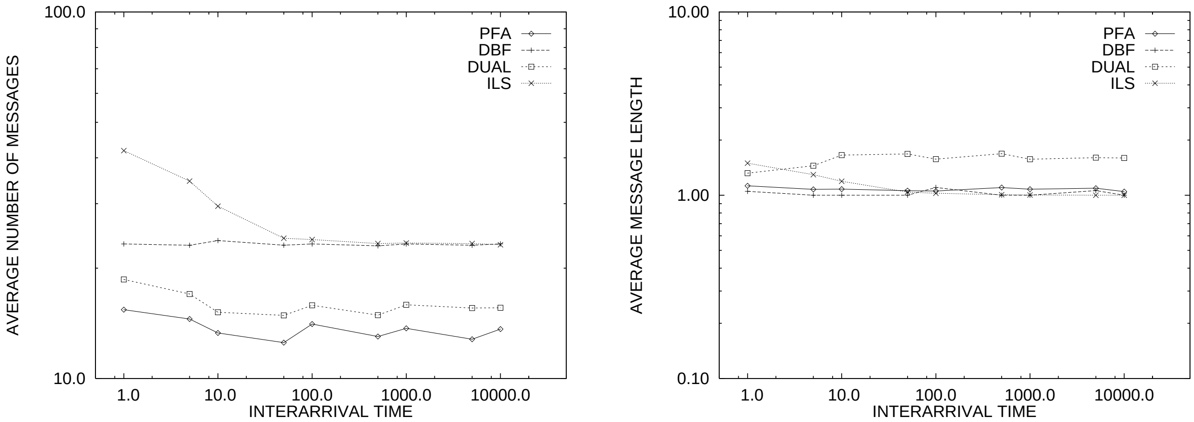

Figure 5.4: Nsfnet

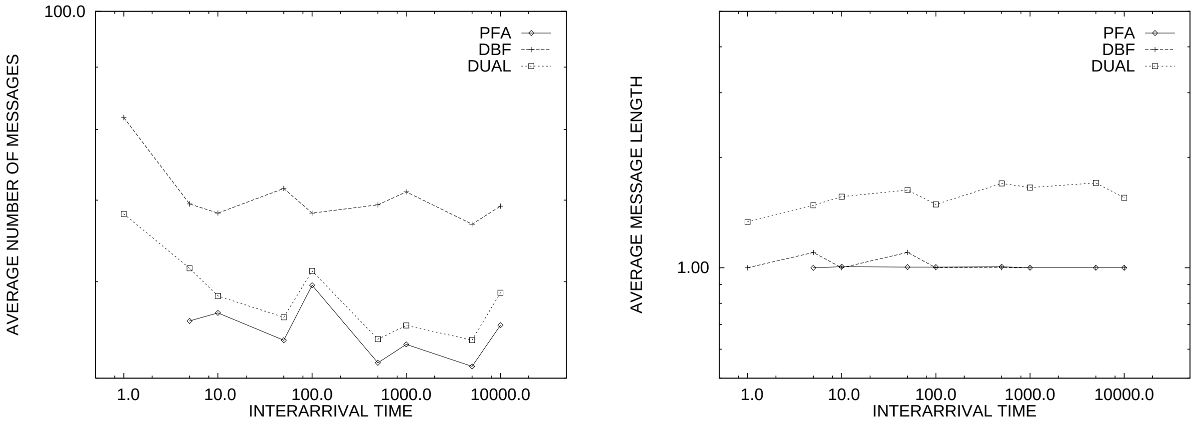

Figure 5.5: ARPANET

In all cases, the average number of messages for DBF and DUAL are more than that of WRP. This results from DBF's counting-to-infinity problem and DUAL's use of an interneighbor coordination mechanism to achieve loop-freedom and this synchronization mechanism spans the entire diameter of the network. ILS sends maximum number of messages since the complete topology information has to be exchanged between neighbors every time the topology changes.

The average length of each message is the highest in DUAL as compared to all other algorithms. The average message length in case of ILS is almost constant since it always sends the complete topology information. Even though we do not have simulation results for ILS in case of Arpanet topology, we can extrapolate the results from the other two network topologies and can expect similar behavior for Arpanet topology also.

5.4 Implementation Status

We have completed an initial implementation of the wireless routing protocol under the BSD version of the UNIX operation system [Ana96]. This implementation contains all features of the basic WRP, and we are currently considering additions to the implementations to support subnetting. Our implementation includes all functions of the basic routing algorithm, the hello protocol, which is used to maintain neighbor discovery information and the reliable exchange of messages (which includes ACKs).

The implementation is mainly done in the user space and the kernel routing tables are updated as and when required. We are also considering the implementation of WRP based on the implementation of some of the existing routing daemons of UNIX such as *routed* and *gated*.

5.4.1 Optimization

In order to reduce the routing control messages in the protocol and thus to minimize the protocol overhead, we limit the number of times a *Hello* packet has to be sent to the neighbors. Any packet received from a neighbor can be treated as an implicit Hello and the hello timer will be reset. Only when there is no traffic in the network, an explicit Hello will be sent to the neighbors. This reduces the number of control messages in the network.

In order to reduce the number of acknowledgments exchanged for each update, we use piggy-backed acknowledgments. After an update is received and before an ACK is sent in response to that update, if the node has any updates to be sent to the neighbor for which it has to send the ACKs, these ACKs will be piggy-backed with the updates.

5.5 Summary

In this chapter, we have described a wireless routing protocol, WRP, which is based on path-finding algorithm. A mechanism has been proposed for the reliable exchange of update messages as part of WRP. The basic algorithm used in WRP has been proved to be correct and WRP's complexity has been analyzed. The performance of the routing algorithm in WRP has been compared with that of an ideal topology broadcast algorithm (ILS), DUAL and DBF for highly dynamic environment through simulations. The simulation results show that WRP provides about 50% improvement in the convergence time as compared to DUAL. The results indicate that WRP is an excellent alternative for routing in wireless networks.

Chapter 6
Congestion-Oriented Routing

The previous chapters described routing algorithms and protocols for various network environments such as hierarchical networks and wireless networks. Using efficient routing algorithms we get small average packet delays but, the network might be able to accept more traffic if flow control is associated with routing. On the other hand, efficient flow control algorithm alone rejects excessive offered load that would necessarily increase the packet delays by saturating network resources. This can possibly be avoided by rerouting packets through non-congested links. Therefore, it is clear that routing and congestion-control are very much interrelated.

In this chapter, we present a congestion-oriented multipath routing algorithm in which we combine congestion-control with routing and present an integrated solution based on credits. Our objective is to provide certain level of performance guarantees in a connectionless network. This we do by defining a two-tier architecture where connection oriented sessions are mapped on to connectionless flows thereby supporting multiple QoS levels. To the end user, this architecture looks very much similar to a connection-oriented architecture through which several levels of performance guarantees can be requested. We first describe the connectionless model and then present the two-tier model.

6.1 Prior Work

One of the drawbacks of the existing Internet routing protocols is that their main routing mechanisms (route computation and packet forwarding) are poorly integrated with congestion control mechanisms. More specifically, today's Internet routing is based on single-path routing algorithms. Even in theory, a routing protocol based on single-path routing is ill suited to cope with congestion. The only thing a single path routing protocol can do to react to congestion is to change the route used to reach a destination. However, as has been documented in [Ber82], allowing a single-path routing algorithm to react to congestion can lead to unstable oscillatory behavior.

Datagram based networks carry traffic that spans a fairly wide range of rates, but their performance is guaranteed on a *best effort* basis only. Recently, quality of service (QoS) has become a very important issue. The newer Internet routing protocols such as IPv6 [DH96] have mechanisms to support different service qualities. This implies that just best-effort service alone does not suffice for datagram networks for future applications. Also, current routing protocols react to congestion *after* the network resources have been wasted i.e., a source reacts to congestion only after it receives a congestion indication from the congested resource in the data path before the source can regulate the rate at which it inputs data into the network. Furthermore, for a connectionless service, any datagram offered to the network is accepted. It is up to the transport protocol to react to congestion *after* network resources are already being wasted. This requires a congestion control scheme which is responsive to the changes in the available bandwidth and efficiently uses the unused bandwidth. The bandwidth must be available on a timely basis allowing application to almost instantaneously use the available bandwidth while maintaining low packet losses.

Hop-by-hop credit-based schemes have a potential for the desired responsiveness [KTA94]. The performance of these schemes is independent of the input traffic pattern. These schemes can be used to fairly share the network bandwidth among competing flows. Also, when there is bandwidth contention, each flow obtains a fair share of the bandwidth.

Our work was motivated by our conjecture that architectural elements similar to those used in a connection-oriented architecture to allow the network to enforce performance guarantees can be used in a connectionless architecture to integrate routing with congestion control, and to provide some delay bounds for the delivery of those datagrams that are accepted in the network. We propose a new framework and protocol for dynamic multipath routing in packet-switched networks that attempts to prevent over-utilization of network resources and hence congestion. Packets are individually routed towards their destinations on a hop-by-hop basis. A packet intended for a given destination is allowed to enter the network if and only if there is at least one path of routers with enough resources to ensure its delivery within a finite time. In contrast to existing connectionless routing schemes, once a packet is accepted into the network, it is delivered to its destination, unless a resource failure prevents it. Each router reserves buffer space for each destination, rather than for each source-destination session as it is customary in a connection-oriented architecture, and forwards a received packet along one of multiple loop-free paths towards the destination. The buffer space and available paths for each destination are updated to adapt to congestion and the dynamic state of the network.

Our framework is based on three main architectural elements namely:

- traffic shaping by means of destination-oriented permit buckets
- traffic separation and scheduling on a per destination basis
- maintenance of dynamic multiple loop-free paths to reduce the delay from source to destination.

Permit buckets consist of permits or tokens fed by periodic updates of credits. To schedule packet transmission, we assume a packet-by-packet generalized processor sharing (PGPS) server [PG93] at each node. To establish loop-free multipaths, we extend prior results on loop-free single-path routing algorithms introduced in [GLAM95]. This results in a congestion-oriented multipath routing architecture that uses a short-term metric based on hop-by-hop credits to reduce congestion over a given link, and a long-term metric based on end-to-end path delay to reduce delay from source to destination. The main contribution of this work is to illustrate the provision of performance guarantees in a connectionless routing architecture.

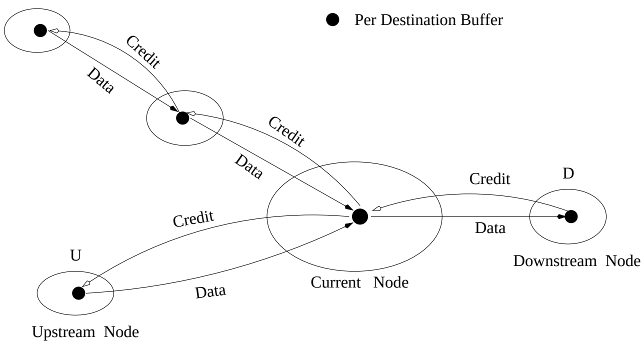

Figure 6.1: Basic Credit-based Scheme

Based on this model of dynamic multipath routing, we present a two-tier architecture for providing several levels of performance guarantees in a connectionless network architecture. This mechanism is capable of supporting multiple QoS for data flows.

6.2 Protocol Description

The protocol can be divided into three functional elements, namely: *packet scheduling and transmission*, *congestion-based credit mechanism* and *maintenance of multiple loop-free paths*. To forward packets to a given destination, the protocol uses two routing metrics: a short-term metric based on hop-by-hop credits to reduce congestion along a link, and a long-term metric based on path-delay to minimize end-to-end delay.

The routing variables associated with each link are determined by periodically monitoring traffic on the incoming and the outgoing links at each node through each neighbor. Given the capacity of each link and the traffic on the link, the utilization of the link can be determined. Credits are reassigned to upstream neighbors contributing to the traffic at a node depending on the relative traffic flow on each of the incoming links. A multipath routing algorithm based on LPA maintains multiple loop-free paths. Each time the network state changes, paths are recomputed and the updated network state is obtained. This is made possible by the periodic exchange of routing information.

Each node maintains a *routing table*, a *distance table*, a *link cost table* and a *link credit table*. The distance table at node i is a matrix that contains, for each destination j and for each neighbor k, the distance (D^i_{jk}) reported to node i by node k regarding destination j, the predecessor reported to node i by node k regarding destination j (p^i_{jk}) and a successor flag ($flag^i_{jk}$) indicating whether neighbor k belongs to the shortest multipath set, for destination j. Node i's routing table is a column vector containing the routing information about the shortest path to all destinations; it maintains information about the distance (D^i_j), predecessor (p^i_j), successor (s^i_j), and the routing parameters (credits and delay) for each shortest path. The neighbor nodes used for packet forwarding from node i to node j are said to belong to the *shortest multipath* from i to j, denoted by SM^i_j. If the neighbor node belongs to SM^i_j, then $flag^i_{jk}$ is set to 1; otherwise it is set to 0. The link cost table maintains the distance information about all the neighboring links and the link-credit table maintains information about the credits available through all the neighboring links for each destination.

6.2.1 Basic Credit-based Mechanism

Figure 6.1 depicts the basic hop-by-hop credit-based flow control mechanism. U is the upstream node and D is the downstream node. A node D is said to be downstream to a node U with respect to destination j if there is a routing path from U to j passing through D. Similarly, node U is an upstream node from D, if D is downstream to U. Two types of packets, data and credits are used. Node U keeps a credit register for each connection (destination) x, which indicates the number of credits available for a connection (destination) x.

Before forwarding the data packets on each link, the sender needs to receive credit information from the credit cells sent by the receiver. At various times, receiver sends credit cells to the sender indicating that there is a certain amount of buffer space available for receiving data packets for a connection (destination). After having received credits, the sender is eligible to forward some number of data packets to the receiver according to the received credit information. Each time sender forwards a data packet, credit count for that connection (destination) is decremented by one.

The advantages of credit-based flow control scheme are maximizing network utilization and controlling congestion.

6.2.2 Message Types

Messages are exchanged among routers for the proper functioning of the protocol. We classify the messages into six types.

Explicit Credit Requests: When a source becomes active and has some data to be sent towards a destination it sends an explicit credit request message to the destination of that flow requesting for credits. Explicit credit requests are associated with sequence numbers. The state of the source indicating the source which initiated the credit request and the corresponding sequence number are recorded at the time of request. (These messages can be viewed analogous to *PATH* messages in RSVP).

Explicit Credit Response: This message is sent in response to an explicit credit request message and indicates the credit availability for data transmission and the sequence number of the request. The available credits are then distributed among the sources that requested for credits. (These messages are analogous to *RESV* messages of RSVP).

Explicit Teardown: This message is sent to reclaim all the credits assigned to a particular flow. A source on recognizing that there are no more packets associated with a flow that needs to be delivered to the destination initiates this message. Credits reserved for a flow are reclaimed at each hop along the way to a destination. (These are similar to *TEARDOWN* messages of RSVP).

Fast Reservation Packet: When a node sends an explicit credit request, its neighbor along the path to a destination responds to this with a fast reservation packet indicating that the source can use small amount of credits to start with before the actual credit allocation is made. This enables sources to start transmitting data without having to wait till it receives credits from the destination and thus prevents starvation. The fast reservation credits are reclaimed when the credits from the destinations are assigned to the flow.

Periodic Credit Updates: Credits are updated periodically to maintain the correct network state information. These are simple update messages which carry the rate at which data packets can be sent and do not have a sequence number associated with it. At every periodic interval, current credit information is exchanged among neighboring nodes and credits are redistributed among active flows at each source accordingly. A credit update which is sent in response to an explicit credit request will be taken into account in the periodic update after the explicit credit response was sent.

Routing Table Update: Whenever the information present in the routing table changes, a routing table update is exchanged between neighbors. These updates are also not associated with sequence numbers. Routing table update updates both the cost and the credit information. The way in which the routing table entries are updated is dependent on the underlying routing algorithm.

6.2.3 Packet Scheduling and Transmission

Packet scheduling is done by means of permit bucket filters for each destination. The *packet-by-packet generalized processor sharing* (PGPS) scheme is used at each server [PG93]. Packets are transmitted as individual entities. A packet is said to have arrived only after the last bit has been received at a node. The server picks up the first packet that would complete service if no additional packets would arrive. Routing is done on a per destination basis over multiple paths.

In a connectionless architecture, the route which a packet takes to reach a destination is determined independently at each hop. All nodes along any path from source to destination has a potential to contribute to the flow for a destination. This necessitates traffic scheduling to be done at each hop to regulate the incoming traffic instead of having intermediate nodes

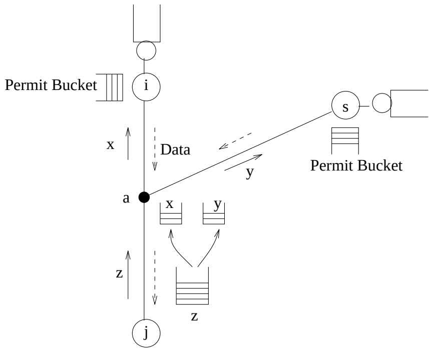

Figure 6.2: Credit Aggregation

as just forwarding nodes as in a connection-oriented architecture [PG94]. PGPS servers are used at each node to regulate the incoming traffic.

The protocol uses two routing metrics for transmitting packets to a given destination: a short-term metric based on hop-by-hop credits to reduce congestion along a link, and a long-term metric based on path-delay to minimize end-to-end delay along the paths. The number of packets sent to a neighbor depends on the credits available through that neighbor. Credits for a destination are sent from a destination towards the source along the reverse paths implied by the routing tables. When a node becomes operational, depending on the availability of resources at each node, credits are distributed among its neighboring nodes. Credit based mechanism is explained in the next section.

The traffic at each node is regulated by permit-buckets, independently for each destination. In the traditional leaky bucket congestion-control scheme, buckets are session oriented i.e., credits are assigned on a per session basis. Data packets accepted from the transmitter and the average rate of flow is controlled by a burst rate for a source-destination session. In our scheme, permit buckets (which are similar to leaky buckets) are destination oriented i.e., at every router permit buckets are maintained for all active destinations. For a given destination j, credits arrive to a node i at a rate ρ^i_j, which is called the *token generation rate* for destination j at node i. The bucket size, denoted by $\sigma^i_j(t)$ [PG93] gives the maximum number of packets that can be transmitted from i to j at time t, and $\sigma^i_j(t)$ is defined for each destination j at time $t \geq 0$ as

$$\sigma_j^i(t) = l_j^i(t) + Q_j^i(t) \tag{6.1}$$

where $l_j^i(t)$ is the number of left-over credits (or tokens) in the bucket at node i for destination j at time t, and $Q_j^i(t)$ is the backlog for destination j at time t. This definition is much the same given in [PG93], the only difference being that here we maintain leaky-bucket parameters for each active destination rather than for each session.

Destination-based credits are aggregated at each node. Each hop is considered as a source; credits sent by the downstream nodes are aggregated at each hop for a given destination and are redistributed among its upstream neighbors. The total available credits at each node for a given destination is the sum of the credits received from its downstream neighbors for that destination. In Figure 6.2, if z is the number of credits received by node a from its downstream neighbors to destination j, then node a maintains a permit bucket of size z. These credits are redistributed among its upstream neighbors relative to the traffic flow along links (i, a) and (s, a) as x and y.

The number of credits left behind, denoted by l_j^i, is the difference in the number of arrivals and the number of credits that arrive within a given time interval. Accordingly, $l_j^i(\tau, t) = A_j^i(\tau, t) - [K_j^i(t) - K_j^i(\tau)]$, where $K_j^i(t)$ is the number of credits that arrive at node i at time t for destination j and $A_j^i(\tau, t)$, the traffic arriving at node i for destination j in the interval $(\tau, t]$ is the number of arrivals in units of credit. The total number of accepted credits in a time period should be less than the credit generation rate. Therefore, with $\tau \leq t$,

$$K_j^i(t) - K_j^i(\tau) \leq \rho_j^i(t - \tau) \tag{6.2}$$

and

$$\sigma_j^i(\tau, t) \geq A_j^i(\tau, t) - \rho_j^i(t - \tau) + Q_j^i(\tau, t) \tag{6.3}$$

Let $t - \Delta t$ be the time when credits were last updated at node i before time t. For the time interval $(t - \Delta t, t)$ we can write Eq. 6.3 as:

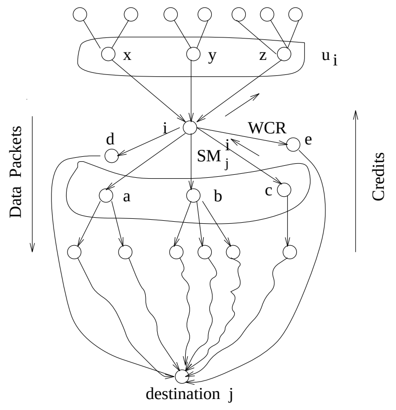

Figure 6.3: Multipath Graph

$$\sigma_j^i(t) \geq A_j^i(t) - \rho_j^i(t) + Q_j^i(t) \tag{6.4}$$

ρ_j^i is related to the number of total available credits at time t and is the sum of the credits available through all the nodes downstream of node i. Consider Figure 6.3, the total number of credits available at router i for destination j is the sum of the credits available from its downstream neighbors a, b, and c for destination j. The total number of packets transmitted to destination j from node i cannot exceed the total available credits at i for j at time t. The number of available credits also depends on the traffic flow on that link which is a measure of the congestion level of that link.

6.2.4 Credit-based Congestion Mechanism

Congestion over a given link is controlled by a hop-by-hop credit-based mechanism. Each node selects a path to a destination based on the bandwidth available through a given link, utilization of that link, and the distance to the destination. The chosen path is subjected to a constraint that the bandwidth available is at least equal to the required bandwidth, and the total bandwidth allocated through a link is less than the capacity of that link. The available bandwidth is then translated into credits. Credits given by a node to its upstream neighbors for a given destination represents the number of packets that the node can accept from its upstream neighbors for that destination. Credits to upstream nodes are sent along the reverse direction of the routing tree. Upstream nodes receive credits from downstream nodes for a given destination before they send data towards the downstream nodes.

Figure 6.4 presents a formal description of the credit allocation scheme. Procedure *Initialize* indicates the action taken by a node when it becomes active for the first time. Procedure *Receive* describes the functions performed by a node when a node receives a periodic update.

Initialization

The number of credits available at each node is determined by the total resources (buffer space) available at that node ($MaxBufs_j$). A part of the total available credits ($Reserve$) is reserved for a *fast reservation mechanism.* A fast reservation mechanism is used to allocate credits to a new source when an explicit credit request is received for a destination. This mechanism sends a minimum number of credits to the new upstream neighbor as explained below. This speeds up the credit allocation process, thus avoiding slow start. Since there is always some reserved credits at each node, nodes upstream do not starve. The reserved credits which are used for fast reservation when the upstream neighbor/ source first initializes are reclaimed when the traffic flow through that path starts. The remaining credits, CR^i_j, are equally distributed among the neighboring nodes, N_i, on startup. This is termed as the weighted credit WCR^i_{jk}. The hop-by-hop delay incurred for the credits to reach next-hop neighbor is incorporated while computing the available credits.

On initialization, credits are equally distributed among neighbors since there will not be any traffic on any of the links of a newly established node. Credits are dynamically assigned thereafter among all the active flows, depending on the traffic flow through each of the links. When node i selects neighbor k as one of its multiple successors to a destination, it sets the successor flag in the update message ($flag^i_{jk}$) to that neighbor to indicate that the neighbor now belongs to shortest multipath set SM^i_j. When node k recognizes this, it includes the node in its set of active neighbors, sends a fixed minimum amount of credit (CR_{min}) to its new neighbor indicating that i can be a possible multipath successor, and redistributes its credits for that destination. This information is communicated to other nodes in the next update interval. The total credits sent to the upstream neighbor is limited by the total available credits at that node. Credit information at each node is updated periodically. This is required since we are not using any reliable mechanism for updating credit information.

Each routing node resets its traffic counters and monitors the incoming and outgoing traffic for all its neighbors. Based on this statistics, the routing parameters ϕ^i_{jk} are computed. The permit bucket parameters are also initialized for each destination. The token generation rate ρ^i_j is initialized to the sum of the credits available through all the neighbors of i to a given destination j in the given time period. The bucket size σ^i_j is initialized to the number of leftover packets since on initialization there is no backlog.

Steady State

Each node monitors the traffic flowing through its incoming and outgoing links periodically and determines the traffic flow on each of its links for all destinations. A periodic update timer is maintained at each router to exchange credit information periodically. The periodic update interval Δt should be at least longer than the maximum round trip time (RTT) delay between two nodes in the network. Each time an update is sent, the timer, $timer^i_j$, is reset (Figure 6.4).

At each node, credits received from all downstream nodes are aggregated and are redistributed to the upstream neighbors. This can be done because the total bandwidth allocated at each link at any given time is no more than the capacity of that link. A node can send data packets to a downstream neighbor only if the credit value through that neighbor is greater than zero. Also, because at each hop credits are distributed based on the traffic flow, the algorithm ensures that information about all active destinations are maintained, i.e., those for which data traffic needs to flow from or through that node. When a new destination for which the bandwidth is not reserved becomes active, or when a node becomes part of the set of loop-free paths to a destination, credits are redistributed using a fast reservation mechanism.

Variables:

p^i_{jl}: credits occupied by packets in transit
q^i_j: credits due to packets already in queue

Procedure Initialize
when router i initializes itself
begin
    $CR^i_j \leftarrow MaxBufs_j - Reserve;$
    do for $k \in N_i$
    begin
        $WCR^i_{jk} \leftarrow \frac{CR^i_j}{|N_i|};$
        $flag^i_{jk} \leftarrow 0;$
    end
    Send credit information to all $x \in N_i$ at next update interval
    Reset $timer^i_j$
end

Procedure Receive(k)
when periodic update is received ($timer^i_j$ expired)
begin
  if $((flag^i_{jk} = 1) \wedge (k \notin SM^i_j))$
    $WCR^i_{jk} \leftarrow CR_{min};\ SM^i_j \leftarrow SM^i_j \cup k;$
  if $((flag^i_{jk} = 0) \wedge (k \in SM^i_j))$
    $WCR^i_{jk} \leftarrow 0;\ SM^i_j \leftarrow SM^i_j - k;$
  do for $m \in N_i$
  begin
    if $m \in SM^i_j\ flag^i_{jm} \leftarrow 1;$
    else $flag^i_{jm} \leftarrow 0;$
  end
  redistribute credits among shortest path neighbors
  $CR^{i'}_j \leftarrow \sum_{l \in SM^i_j} WCR^i_{jl} - p^i_{jl};$
  $CR^i_j \leftarrow CR^{i'}_j - q^i_j;$
  $WCR^i_{jk} \leftarrow CR^i_j \times \phi^i_{jk} \mid i \in SM^k_j$
  Send credit information to all $x \in N_i$ at next update interval
  Reset $timer^i_j$
end

Figure 6.4: Credit Distribution Mechanism

Explicit Credit Request and Response

When a new source becomes active and the source needs credits to transfer data, an explicit credit request is sent to all neighbors in the shortest multipath set. This is responded by a credit response. Also, the initial credit assignment is done using fast-reservation mechanism.

Explicit credit response indicates the amount of credits allocated by the destination through multiple paths for the flow requested.

Explicit Teardown

A teardown message reclaims the credits allocated to a flow. Each teardown message is associated with a sequence number representing the flow. Although it is not necessary to explicitly reclaim the credits of a flow, it is recommended that all sources send explicit teardown message as soon as a session is completed.

Teardown message is initiated by a source and is forwarded hop-by-hop all the way to the destination. The state of a node is updated along the way and the credits are reclaimed. These messages are not delivered reliably. The loss of a teardown message will not cause a protocol failure because the unused credits will eventually time out and are reclaimed.

One-Pass Reservation Mechanism

To ensure that there is no starvation in the protocol, a fast reservation mechanism is used. This enables the downstream nodes to respond immediately without having to wait till the nodes hear from their destinations about the credit availability. This scheme is similar to one-pass reservation mechanism of RSVP [ZDE+93].

Sources send explicit credit requests to a destination requesting credits for data transmission. On receiving these requests, credits are allocated for that session; source and all the nodes along the way are informed about this. Credits indicate the rate at which a source can send data to respective destinations.

Each node maintains a soft-state information about the network state. The state information is periodically refreshed by periodic credit updates. If a credit update is not received before certain time interval, those credits are reclaimed. At each periodic timeout interval, the state of the network is updated. The soft-state information has the potential to change every periodic time interval.

Periodic Updates

Credit (resource reservation) information is updated periodically. The allocated resources are reclaimed after a certain time if the state information is not refreshed. The timer value for updating the state information is set at each hop independently. Floyd and Jacobson [FJ94] have shown that periodic updates generated by independent network nodes can become synchronized. This can lead to disruption in network services as the periodic messages contend with other network traffic for link and forwarding resources. Therefore, periodic credit update messages must avoid synchronization and ensure that any synchronization that may occur is not stable.

Because of this, the refresh timer should be randomly set to a value in the range [0.5R, 1.5R] where R is the periodic update timer used to generate updates. The value R is chosen locally at each node. A smaller R speeds up the adaptation to network state changes but increases the protocol overhead. A node may therefore adjust the effective R dynamically to control the amount of overhead due to periodic update messages. The default value of R is set to 30 seconds

The end-to-end delay associated with packets to each destination are also estimated periodically. If the measured delay does not satisfy the required QoS, that successor will no longer be selected as a feasible successor to that destination and this information is communicated to all the neighboring nodes. It then determines the total available credits for a given destination j and the credits are redistributed among its upstream neighbors after reserving a fraction of the credits for the initialization phase. The philosophy behind this fast reservation mechanism is similar to a fast bandwidth reservation scheme in which, the data transmission begins before a connection has been completely established.

Figure 6.5 shows the distance table at node i for destination j for the configuration in Figure 6.3. The flag field indicates whether the neighbor belongs to the shortest multipath set or not. The distance gives the sum of the link costs along the path to destination j and credits gives the number of available credits through that path. A credit of 0 implies that packets cannot be forwarded through that path.

Destination	Neighbor	Flag	Distance	Credits
j	a	1	a1	a2
j	b	1	b1	b2
j	c	1	c1	c2
j	d	0	d1	0
j	e	0	e1	0
j	x	0	x1	0
j	y	0	y1	0
j	z	0	z1	0

Figure 6.5: Distance Table at node i for destination j

The number of credits available at a node is determined by the flow on its links and the total traffic seen by that node. If f^k_{ji} is the incoming flow on link (k, i) to destination j as seen by node i, and r^i_j is the traffic originated at i for destination j, we define the total input traffic seen by i for destination j as the sum of all the incoming traffic at node i, and denote it by t^i_j. Furthermore, by the conservation of flow, the sum of all the traffic arriving at a node must be equal to the sum of all the traffic departing from a node for each destination j. Therefore, for destination j, the total incoming flow is equal to the total outgoing flow at node i, and

$$t^i_j = \sum_{k \ | \ i \in SM^k_j} [f^k_{ji}] + r^i_j = \sum_{m \ \in \ SM^i_j} f^i_{jm} \tag{6.5}$$

For convenience, a routing variable, denoted by ϕ^i_{jk}, is defined for each link (i, k) as the ratio of the flow on each link with respect to the total flow on all outgoing links for a given destination j. From Eq. 6.5 and with N_i denoting the neighbor set of i, we have:

$$\phi^i_{jk} = \frac{f^i_{jk}}{t^i_j} \quad \forall k \in N_i \tag{6.6}$$

Because node i itself can also contribute to the total traffic, by the conservation of flow, it must be true that

$$\sum_{k \in SM^i_j} \phi^i_{jk} \ \leq \ 1 \tag{6.7}$$

The distribution of credits to upstream neighbors depends on the traffic flow on that link, which in turn depends on the routing variable ϕ^i_j associated with that link. The number of credits a node sends to an upstream neighbor is called the weighted credit (WCR). Credits are weighted by the traffic flow on a given link. The token generation rate for a given update period Δt can now be defined as

$$\rho^i_j(t) = \frac{Credits\ \ available\ in\ update\ period\ before\ t}{\Delta t}$$

$$\rho^i_j(t) = \frac{\sum_{k=1}^{SM^i_j}\ WCR^i_{jk}(t)}{\Delta t} \tag{6.8}$$

To obtain a correct estimate of the credits available at each node at any given time, we need to take into account the delay associated with the propagation of credits. This can be done either by estimating the credits available as in [KTA94] or by explicitly sending a marker. We opt for the estimation mechanism. Credits are sent to the immediate upstream neighbor, i.e., they propagate only one hop. The update period used for updating routing information is considered as one round-trip delay by a data packet. Therefore, to obtain a correct estimate of the available credits at a node, we have to take into account the data packets that a sender has already forwarded over the link in the previous round-trip time (RTT) and the data packets that are already queued from the past RTT. Therefore, the total available credits at node i for a destination j, denoted by CR^i_j, is the difference between the sum of all the weighted credits available from its downstream neighbors d_i (equivalently, sum of all the credits on its outgoing links) belonging to the shortest multipath and the credits which are already being used, i.e.,

$$CR^i_j = \sum_{l\ \in\ SM^i_j} [WCR^i_{jl} - p^i_{jl}] - q^i_j \tag{6.9}$$

where WCR^i_{jl} is the weighted credit obtained from the downstream neighbor l over an update period (which depends on the flow on the link (i,l)), p^i_{jl} is the number of credits occupied by the packets that are already in transit on link (i, l), and q_j^i is the number of credits due to the data packets that are already in the queue at node i for destination j which were not completely transmitted since the previous update period. If a node does not have credits for a given destination j, then CR_j^i is set to zero.

6.2.5 Correctness of Credit-based Scheme

The correctness of the credit based mechanism (i.e., showing that it has no deadlocks and that packets are not dropped) can be proven in a similar way as for virtual-circuit connections [OSV94]. For the purposes of such a proof, we make the following assumptions.

- protocol is initialized properly i.e., all nodes have correct credit and routing information
- there are no link errors or link and node failures (steady state)

Lemma 6.1 *The congestion-oriented credit mechanism never drops packets accepted into the network.*

Proof: The total credits available at a node is given by:

$$CR_j^i \geq \sum_{l \in SM_j^i} [WCR_{jl}^i - p_{jl}^i] - q_j^i$$

Also, the total available credits is less than the maximum buffer space at a node.

$$MaxBuffers_j \geq CR_j^i$$

This is true since we reserve some buffer space (credits) for the fast reservation mechanism.

At any node, the number of credits available for a given destination j is at least equal to the packets sent downstream for that destination. This is true since a packet can not be forwarded unless we have a credit to do so. If a packet is in flight from a sender to a receiver, then $p_{jl}^i = 0$ and $CR_j^i < MaxBuffers_j$. Thus, the received packet will not be dropped as there is at least one empty buffer that can be used.

This proves the Lemma. □

Lemma 6.2 *The credit based mechanism is starvation free.*

Proof: To prove this Lemma, we consider three cases – (a) initialization (b) when the nodes are active and (c) when a explicit credit request is received.

Case (a): On initialization, when a new node comes up, the node distributes its credits equally among its neighbors and receives a minimum number of non-zero credits from its neighbors indicating the presence of a possible path to a destination through that neighbor. Therefore, on startup since always a minimum number of credits are available to the new node, the credit mechanism is starvation free.

Case (b): When a new node becomes a member of the shortest multipath set (SM_j^i), the successor flag will be set in the update message. On receiving this update, the neighbor sends a minimum number of credits initially using the reserved bandwidth available through the fast reservation mechanism and later on dynamically updates the credits depending on the traffic on the link.

Case (c): An explicit request for credits from a source is received by means of a routing table update. This update sets the successor flag in the update message. Therefore, the message will be treated as any other routing table update and is processed similar to case (b) if the router has credits to send to its upstream neighbors. Otherwise, the router sends routing table updates to its shortest multipath neighbors (downstream neighbors) requesting for credits to send packets to the destination. In the worst-case, this process can continue till the routing table update reaches the destination and will terminate at the destination.

This ensures that a node will always have credits when it requires to forward a packet. Therefore, the protocol is starvation free. □

Theorem 7 *In the congestion-based credit mechanism, if there is a packet at a node to be sent, it will be eventually sent.*

Proof: From Lemma 6.1, packets are not dropped. Therefore, packet arriving at a node will either be queued or is forwarded. From Lemma 6.2, whenever a node needs to forward a packet, it always has the credits to do so.

Therefore, packets arriving at a node will be always sent eventually. Of course, when links or nodes fail, packets accepted in the network may have to be dropped. The same occurs in a connection-oriented architecture when resources fail along a connection already established. This proves the theorem □

6.2.6 Maintenance of Loop-Free Multipaths

The primary objective of maintaining multiple loop-free paths is to minimize the end-to-end path delay by reducing network congestion along the path. The distance reported by neighbor k to node i for destination j is denoted by D^i_{jk} and node i's distance to its neighbor k is denoted by d_{ik}. The distance to neighbor k is the sum of the propagation delay δ^i_k and the per hop packet delay through neighbor k, d^i_{jk}. i.e., $d_{ik} = d^i_{jk} + \delta^i_k$. The path delay at node i along node k at a given time t, denoted by $\breve{D}^i_{jk}$ is $\breve{D}^i_{jk} = D^i_{jk} + d_{ik}$.

The *shortest multipath set* of i for destination j (SM^i_j) is a set of neighbors of i that provide loop-free paths to j. The delay at node i to destination j at time t is computed as the weighted average path delay through all the nodes in the shortest multipath at node i; it is denoted by $D^i_j(t)$. This delay is weighted by the fraction of the traffic going through that path, i.e.,

$$D^i_j(t) = \sum_{k \in SM^i_j(t)} \phi^i_{jk}(t).\breve{D}^i_{jk}(t) = \sum_{k \in SM^i_j(t)} \frac{f^i_{jk}(t)}{t^i_j(t)}.\breve{D}^i_{jk}(t) \tag{6.10}$$

The flow from i to each neighbor in SM^i_j depends on the credits available through that neighbor. Assuming that packets are of fixed size and that each packet corresponds to one credit, we can say that a packet flows on a link if at least one credit is available on that link. This implies that the number of packets that can flow on a link is equal to the number of credits available through that link; therefore,

$$D_j^i(t) = \frac{1}{t_j^i(t)} \sum_{k \in SM_j^i(t)} [WCR_{jk}^i(t).\breve{D}_{jk}^i(t)] \tag{6.11}$$

If the packets are of variable lengths, the packet length is a multiple of credits and for simplicity we can assume that on an average, each packet requires the same number of credits C. With this simplification, the total number of packets transmitted along a link with WCR_{jk}^i credits is a constant K times the total available credits; therefore,

$$D_j^i(t) = \frac{K}{t_j^i(t)}. \sum_{l \in SM_j^i(t)} WCR_{jl}^i(t).\breve{D}_{jl}^i(t) \tag{6.12}$$

Multiple loop-free paths from each node to a destination are maintained by means of a shortest multipath routing algorithm (SMRA), which is based on LPA [GLAM95]. As explained earlier, LPA belongs to the class of path-finding algorithms, which achieves loop-freedom at every instant using single-hop interneighbor coordination mechanism. Any change in distance is notified by event-driven update messages. An update message from router i consists of a vector of entries; each entry specifies a destination j, an update flag, a successor flag, the reported distance to that destination and the reported credits available to destination j through that neighbor. The update flag indicates whether the entry is an update ($u_j^i = 0$), a query ($u_j^i = 1$) or a reply to a query ($u_j^i = 2$).

A detailed specification of SMRA is given in Figures 6.6 and 6.7. Procedures *Init1* and *Init2* are used for initialization. Procedure *Message* is executed when a router processes an update message, a query or a reply; procedures *LinkUp*, *LinkDown* and *Change* are executed when a router detects a new link, link failure or a change in the link cost respectively. We refer to these as the event handling procedures. An update, query or a reply are handled by procedures *Update*, *Query* and *Reply* respectively. Procedure *Update Timer* updates the periodic update timer and updates the credit information. Procedures *DT_Update* and *RT_Update* updates the distance and the routing tables respectively.

A router i can be active or passive for destination j at any given time. Node i is active for destination j if it does not have a feasible successor to destination j and is waiting for at least one reply from a neighbor; node i is passive otherwise. A router i initializes itself

```
Procedure Init1
when router $i$ initializes itself
do begin
    set a link-state table with
    costs of adjacent links;
    $N \leftarrow \{i\}$; $N_i \leftarrow \{x \mid d_{ix} < \infty\}$;
    for each ($x \in N_i$)
    do begin
        $N \leftarrow N \cup x$; $tag_x \leftarrow$ null;
        $s^i_x \leftarrow$ null; $p^i_x \leftarrow$ null;
        $D^i_x \leftarrow \infty$; $FD^i_x \leftarrow \infty$
    end
    $s^i_i \leftarrow i$; $p^i_i \leftarrow i$; $tag^i_i \leftarrow$ correct;
    $D^i_i \leftarrow 0$; $FD^i_i \leftarrow 0$;
    for each $j \in N$ call Init2($x$, $j$);
    for each ($n \in N_i$) do
    add (0, $i$, 0, $i$) to $LIST_i(n)$;
    call Send
end

Procedure Init2($x$, $j$)
begin
    $D^i_{jx} \leftarrow \infty$; $p^i_{jx} \leftarrow null$;
    $s^i_{jx} \leftarrow null$; $r^i_{jx} \leftarrow 0$;
end

Procedure Send
begin
    for each ($n \in N_i$)
    do begin
        if ($LIST_i(n)$ is not empty)
        then send message with
          $LIST_i(n)$ to $n$
        empty $LIST_i(n)$
    end
end

Procedure Reply($j$, $k$)
begin
    $r^i_{jk} \leftarrow 0$;
    if ($r^i_{jn} = 0, \forall n \in N_i$)
    then if (($\exists x \in N_i \mid D^i_{jx} < \infty$)
      or ($D^i_j < \infty$))
        then call Passive_Update($j$)
    else call Active_Update($j$, $k$)
end

Procedure Message
when router $i$ receives a message
  on link ($i$, $k$)
begin
    for each entry ($u^k_j, j, RD^k_j, rp^k_j$)
      such that $j \neq i$
    do begin
        if ($j \notin N$)
        then begin
            if ($RD^k_j = \infty$)
            then delete entry
            else begin
                $N \leftarrow N \cup \{j\}$; $FD^i_j = \infty$;
                for each $x \in N_i$
                  call Init2($x$, $j$)
                $tag^i_j \leftarrow null$;
                call DT_Update($j, k$)
            end
        end
        else
            $tag^i_j \leftarrow null$; call DT_Update($j, k$)
    end
    for each entry ($u^k_j, j, RD^k_j, rp^k_j$) left
    such that $j \neq i$
    do case of value of $u^i_j$
        0:  [Entry is an update]
            call Update($j$, $k$)
        1:  [Entry is a query]
            call Query($j$, $k$)
        2:  [Entry is a reply]
            call Reply($j$, $k$)
    end
    call Send
end

Procedure Update($j$, $k$)
begin
    if ($r^i_{jx} = 0, \forall x \in N_i$)
    then begin
        if (($s^i_j = k$) or ($D^i_{jk} < D^i_j$))
        then call Passive_Update($j$)
    end
    else call Active_Update($j$, $k$)
end

Procedure Passive_Update($j$)
begin
    $DT_{min} \leftarrow Min\{D^i_{jx}\ \forall\ x \in N_i\}$;
    $FCSET \leftarrow \{n \mid n \in N_i,\ D^i_{jn} = DT_{min},$
      $D^n_j < FD^i_j\}$;
    if ($FCSET \neq \emptyset$) then begin
      call RT_Update($j$, $DT_{min}$);
      $FD^i_j \leftarrow Min\{D^i_j, FD^i_j\}$
    end
    else begin
        $FD^i_j = \infty$; $r^i_{jx} = 1, \quad \forall x \in N_i$;
        $D^i_j = D^i_{j\, s^i_j}$;
        $p^i_j = p^i_{j\, s^i_j}$;
        if ($D^i_j = \infty$) then $s^i_j \leftarrow$ null;
        $\forall$ x $\in N_i$ do begin
          if (query and x = k)
          then $r^i_{jk} \leftarrow 0$;
          else add (1, $j$, $\infty$, null)
            to $LIST_i(x)$
        end
    end
end

Procedure Query($j$, $k$)
begin
    if ($r^i_{jx} = 0 \forall x \in N_i$)
    then begin
        if ($D^i_j = \infty$ and $D^i_{jk} = \infty$)
        then add (2, $j$, $D^i_j$, $p^i_j$)
          to $LIST_i(k)$
        else begin
          call Passive_Update($j$);
          add (2, $j$, $D^i_j$, $p^i_j$)
            to $LIST_i(k)$;
        end
    else call Active_Update($j$, $k$)
end
```

Figure 6.6: SMRA Specification

in the passive state with an infinite distance to all its known neighbors and a zero distance to itself. The maximum allowable distance to reach neighbor, defined below, is also set to ∞. Routers send updates containing distance and credit information for themselves to all their neighbors. When the destinations become operational, routers inform their neighbors about the available credits to all other nodes.

Each routing update updates the cost and the credit information. Cost and credit information are exchanged among neighbors when the state of the network changes while credit information is updated periodically also. An update can contain full routing table or increments of the routing table in different update messages. After initialization, only incremental updates are sent.

```
Procedure Link_Up (i, k, d_ik)
when link (i, k) comes up do begin
    d_ik ← cost of new link;
    if (k ∉ N) then begin
        N ← N ∪ {k}; tag^i_k ← null;
        D^i_k ← ∞; FD^i_k ← ∞;
        p^i_k ← null; s^i_k ← null;
        for each x ∈ N_i do call Init2(x, k)
    end
    N_i ← N_i ∪ {k};
    for each j ∈ N do call Init2(k, j);
    for each j ∈ N − k | D^i_j < ∞ do
        add (0, j, D^i_j, p^i_j) to LIST_i(k);
    call Send
end

Procedure Link_Down(i, k)
when link (i, k) fails do begin
    d_ik ← ∞;
    for each j ∈ N do begin
        call DT(j, k);
        if (k = s^i_j) then tag^i_j ← null
    end
    delete column for k in distance table;
    N_i ← N_i − {k};
    delete r^i_jk;
    for each j ∈ (N − i) | k = s^i_j
        call Update(j, k)
    call Send
end

Procedure DT_Update(j, k)
begin
    D^i_jk ← RD^k_j + d_ik; p^i_jk ← rp^k_j;
    for each neighbor b do begin
        h ← j;
        while (h ≠ i or k or b) do h ← p^b_h;
        if (h = k) then begin
            D^i_jb ← D^i_kb + RD^k_j; p^i_jb ← rp^k_j;
        end
        if (h = i) then begin
            D^i_jb ← ∞; p^i_jb ← null;
        end
    end
end

Procedure Update_Timer:
when the periodic update timer expires
begin
    reset timer;
    update credit information;
    for all i ∈ N_i
        send periodic update;
end

Procedure Active_Update(j, k)
begin
    if (k = s^i_j) then begin
        D^i_j ← D^i_jk; p^i_j ← p^i_jk;
    end
end
Procedure Link_Change (i, k, d_ik)
when d_ik changes value do begin
    old ← d_ik;
    d_ik ← new link cost;
    for each j ∈ N do begin
        call DT_Update(j, k);
        for each j ∈ N
        do if (D^i_j > D^i_jk or k = s^i_j)
        then tag^i_j ← null;
    end
    for each j ∈ N do begin
        if (d_ik < old)
        then for each j ∈ N − i | D^i_j > D^i_jk
            do call Update(j, k)
        else for each j ∈ N − i | k = s^i_j
            do call Update(j, k)
    end
    call Send
end
Procedure RT_Update(j, DT_min)
begin
    if (D^i_{j s^i_j} = DT_min)
    then ns ← s^i_j;
    else ns ← b | {b ∈ N_i and D^i_jb = DT_min};
    x ← j;
    while (D^i_{x ns} = Min{D^i_xb ∀ b ∈ N_i}
     and (D^i_{x ns} < ∞) and (tag^i_x = null))
    do x ← p^i_{x ns};
    if (p^i_{x ns} = i or tag^i_x = correct)
    then tag^i_j ← correct;
    else tag^i_j ← error;
    if (tag^i_j = correct)
    then begin
        if (D^i_j ≠ DT_min or p^i_j ≠ p^i_{j ns})
        then add (0, j, DT_min, p^i_{j ns})
          to LIST_i(x)  ∀x ∈ N_i;
        D^i_j ← DT_min; p^i_j ← p^i_{j ns};
        s^i_j ← ns;
    end
    else begin
        if (D^i_j < ∞)
        then add (0, j, ∞, null)
            to LIST_i(x)  ∀x ∈ N_i;
        D^i_j ← ∞; p^i_j ← null;
        s^i_j ← null;
    end
end
```

Figure 6.7: SMRA Specification (cont...)

For a given destination, a router updates its routing table differently depending on whether it is *passive* or *active* for that destination. A router that is passive for a given destination can update the routing-table entry for that destination independently of any other routers, and simply chooses as its new distance to the destination to be the shortest distance to that destination among all neighbors, and as its new feasible successor to that destination to be any neighbor through whom the shortest distance is achieved. In contrast, a router that is or becomes active for a given destination must synchronize the updating of its routing-table entry with other routers.

When a router is passive and needs to update its routing table for a given destination j after it processes an update message from a neighbor or detects a change in the cost or availability of a link or a change in the credit information, it tries to obtain a *feasible successor*. From router i's standpoint, a feasible successor toward destination j is a neighbor router k that satisfies the maximum allowable distance condition (MADC) given by the following two equations [GLAM95]:

$$\begin{aligned} D_j^i &= D_{jk}^i + d_{ik} = Min\{D_{jp}^i + d_{ip} \mid p \in N_i\} \\ D_{jk}^i &< MAD_j^i \end{aligned} \tag{6.13}$$

where MAD_j^i is the *maximum allowable distance* for destination j, and is equal to the minimum value obtained for D_j^i since the last time router i transitioned from active to passive state for destination j. Router i adjusts MAD_j^i depending on the congestion level of the network.

If router i finds a feasible successor, it remains passive and updates its routing-table entry as in the Distributed Bellman-Ford algorithm [BG92]. Alternatively, if router i cannot find a feasible successor, it first sets its distance equal to the addition of the distance reported by its current successor and the cost of the link to that neighbor. The router also sets its maximum allowable distance equal to its new distance. After updating its tables, a router becomes active by sending a query in an update message to all its neighbors; such a query specifies the router's new distance through its current successor. It then sets the destination's reply-status table entry for each link to one, indicating that it expects a reply from each neighbor for that destination.

Once active for destination j, router i cannot change its feasible successor, MAD_j^i, the value of the distance it reports to its neighbors, or its entry in the routing table, until it receives all the replies to its query. A reply received from a neighbor indicates that such a neighbor has processed the query and has either obtained a feasible successor to the destination, or determined that it cannot reach the destination. After node i obtains all the replies to its query, it computes a new distance and successor to destination j, updates its feasible distance to equal its new distance, and sends an update to all its neighbors.

Multiple changes in link cost or availability are handled by ensuring that a given node is waiting to complete the processing of at most one query at any given time. The mechanism used to accomplish this is specified in [GLAM95], and is such that a node can be either passive or in active state, and it processes any pending update or distance increases that occurred while it was active.

Ensuring that updates will not be sent in the network when some destination is unreachable is easily done. If node i has set $D^i_j = \infty$ already and receives an input event (a change in cost or status of link $(i,\ k)$, or an update or query from node k) such that $D^i_{jk} + d_{ik} = \infty$, then node i simply updates D^i_{jk} or d_{ik}, and sends a reply to node k with $RD^i_j = \infty$ if the input event is a query from node k. When an active node i has an infinite maximum allowable distance and receives all the replies to its query such that every neighbor offers an infinite distance to the destination, the node simply becomes passive with an infinite distance.

When node i establishes a link with a neighbor k, it updates the value of d_{ik} and assumes that node k has reported infinite distances to all destinations and has replied to any query for which node i is active. Furthermore, if node k is a previously unknown destination, node i sets $s^i_k = null$, and $D^i_k = RD^i_k = MAD^i_k = \infty$. Node i also sends to its new neighbor k an update for each destination for which it has a finite distance.

When node i is passive and detects that link $(i,\ k)$ has failed, it sets $d_{ik} = \infty$ and $\breve{D}^i_{jk} = \infty$. After that, node i carries out the same steps used for the reception of a link-cost change in the passive state.

Because a router can become active in only one interneighbor coordination mechanism per destination at a time, it can expect at most one reply from each neighbor. Accordingly,

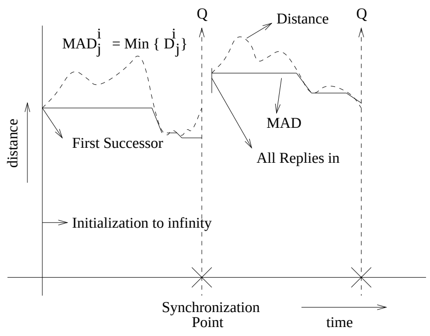

Figure 6.8: Maximum Allowable Distance Condition

when an active node i loses connectivity with a neighbor n, node i can set $r^i_{jn} = 0$ and $D^i_{jn} = \infty$, i.e., assume that its neighbor n has sent any required reply reporting an infinite distance. When node i becomes passive again, it must find a neighbor that satisfies the MADC using the value of MAD^i_j set at the time node i became active in the first place. After finding a new successor, the permit bucket parameters ρ^i_j and σ^i_j are also updated.

Figure 6.8 gives a graphical representation of how MAD is updated. The point at which a new diffusing computation starts is a *synchronization point*. It can be noted that between two synchronization points the value of MAD can only decrease or remain the same. This ensures that the algorithm is loop-free.

To route packets to a destination j, each router uses the following rule to select the neighbor routers that should belong to its shortest multipaths for j:

Shortest Multipath Condition (SMC): At time t, router i can make node $k \in N_i(t)$ part of SM^i_j if and only if $D^i_{jk}(t) < MAD^i_j(t)$.

When nodes choose their successors using SMC, the path from source to destination obtained as a result of this is loop free at every instant. The proof of correctness and loop-freedom of SMRA is basically the same as that provided in [GLAM95] for LPA.

6.3 Worst-Case Steady-State Delay

In this section, we derive an upper bound on the end-to-end steady-state path delay from node i to destination j (D^{i*}_j) as a function of the credits available through each path under steady state. Steady-state means that all distances and credit information is correct at every router. This bound demonstrates that it is possible to provide performance guarantees in a connectionless routing architecture. The delay experienced by a packet accepted into the network is the time required by a data packet to reach its destination router from a source. This includes both the propagation delay and the queueing delay. Path delay can also be interpreted as the time it would take for a destination j backlog to clear when there are no more arrivals after time t.

Parekh and Gallager have analyzed worst-case session delay in a connection-oriented network architecture [PG94]. We adopt a similar approach for each destination in a connectionless architecture. To do this, we assume a stable topology in which all routers have finite distances to each other. We also make use of the fact that SMRA enforces loop-freedom at every instant on all paths in the shortest multipath sets.

In a connectionless network where routes are computed distributedly, the path taken by a packet can change dynamically depending on the congestion level in the network. Routing is done on a hop-by-hop basis, independently at each router. Therefore, the total traffic at a node will be the sum of the traffic on all its links connecting to upstream neighbors. To obtain an expression for the worst-case bound, we make the following assumptions:

1. Each node sends traffic to destination j as long as credits are available (non-zero) for that destination along any of its chosen paths.

2. At every node m, traffic for every destination is treated independently.

3. Traffic arriving at a node i for destination j in the interval $(0, t)$ (denoted by A^i_j) is the sum of the traffic from all its upstream neighbors to destination j and the traffic originated at the node i itself, denoted by $r^i_j(t)$, i.e.,

$$\begin{aligned} A^i_j(t) &= r^i_j(t) + \sum_{n, i \in SM^n_j(t)} f^n_{ji}(t) && (6.14) \\ &= r^i_j(t) + \sum_{l | i \in SM^l_j(t)} \phi^i_{jl}(t).A^i_{jl}(t) && (6.15) \end{aligned}$$

Each router in a connectionless network can itself be a source to any given destination. At each node, traffic to destination j is constrained by a permit bucket filter. The worst-case delay and backlog is upper bounded by an additive scheme due to Cruz [Cru91]. The rate at which the packets are serviced at each node depends on the permit bucket or leaky bucket parameters σ_j^i and ρ_j^i for a given destination j. The parameter σ_j^i gives the permit bucket size and ρ_j^i the credit generation rate at node i. Therefore, the number of packets that are being serviced at a node is a function of σ_j^i and ρ_j^i.

The minimum service rate g_{jm}^i at any node i is the fraction of the input traffic at node i for destination j. The fraction of the traffic is determined by the ratio of the routing variables of the links, which is a function of the traffic flow; Therefore,

$$g_{jm}^i = \frac{\phi_{jm}^i}{\sum_{k=1}^{SM_j^i(t)} \phi_{jk}^i} t_j^i \tag{6.16}$$

The minimum clearing rate of a given path is $g_l^{'} = \min_{m \in P(i,j)} g_{jm}^i$. When $g_l^{'} > \rho_j^l$, the system with respect to destination j is said to be locally stable. The input traffic rate at node i to destination j is the sum of all the incoming traffic destined for j for which i is the intermediate node and the traffic originated at i itself (Eq. 6.5). With these constraints, the bound on the delay for a given destination can be obtained using a similar approach as in [PG94].

The delay on a link (i,k) (per hop delay) d_{jk}^i for a given destination j is the sum of the queueing delay and the propagation delay on that link. The link propagation delay (δ_k^i) depends on the congestion level of the link as well as the link capacity. Propagation delay is defined as the time taken for a packet to reach a destination from a source. Every packet is time-stamped when it leaves a node and the time at which the packet reaches the neighbor is noted. The difference between the two gives a one-hop delay. The average of this delay over a given period of time gives the propagation delay δ_k^i.

The queueing delay is the time a packet has to wait at a node before it is processed. The waiting time of a packet depends on the number of packets already present in the queue at the time a packet arrives. This is referred to as the *backlog* at node i for destination j and is denoted by Q^i_j. Therefore, the delay on link (i,l) for destination j at time t is

$$d^i_{jl}(t) = \delta^i_l(t) + Q^i_{jl}(t).\delta^i_l(t) = \delta^i_j(t)[1 + Q^i_{jl}(t)] \tag{6.17}$$

The backlog number of packets for a given destination j at a given time t can be defined as the difference in the incoming and the outgoing traffic at a node, i.e.,

$$Q^i_j(t) = A^i_j(t) - S^i_j(t) \tag{6.18}$$

This takes into account both the processing delay and the queueing delay experienced at each hop. For every interval $(\tau, t]$,

$$S^i_j(\tau, t) \geq (t - \tau)g^{'}_i \tag{6.19}$$

If the minimum clearing time $g^{'}_i$ is greater than the token generation rate ρ^i_j for a given destination, we can obtain a bound on the backlog and hence the path delay. Let $\tau < t$ be the time at which there are no backlogged packets in the network. Then, because $g^i_j \geq \rho^i_j$ and all the destinations are permit bucket constrained,

$$S^i_j(\tau, t) \geq (t - \tau)\rho^i_j(t - \tau) \tag{6.20}$$

6.3.1 Negligible Packet Size

We first obtain a bound on end-to-end path delay assuming that the size of the packet may not contribute significantly to the delay component. The arrivals at each node i is the sum of the arrivals at all the upstream nodes for destination j and the traffic originated at node i itself. For all $t \geq \tau \geq 0$ we have,

$$A^i_j(\tau, t) = r^i_j(\tau, t) + \sum_{l|i \in SM^l_j(t)} \phi^i_{jl}(\tau, t).A^l_j(\tau, t) \tag{6.21}$$

The maximum backlog traffic Q_j^{i*} for destination j is the difference between the arrivals in the interval $(\tau, t]$ and the total packets serviced in the same interval at node i. For $\phi_{jl}^i > 0$,

$$
\begin{aligned}
Q_j^{i*}(\tau,t) &\leq A_j^i(\tau,t) - S_j^i(\tau,t) && (6.22)\\
Q_j^{i*}(\tau,t) &\leq r_j^i(\tau,t) - S_j^i(\tau,t) \\
&\quad + \sum_{l|i\in SM_j^l(t)} [\phi_{jl}^i(\tau,t).A_j^l(\tau,t)] && (6.23)\\
Q_j^{i*}(\tau,t) &\leq [r_j^i(t) - r_j^i(\tau)] - S_j^i(\tau,t) \\
&\quad + \sum_{l|i\in SM_j^l(t)} [\phi_{jl}^i(\tau,t).A_j^l(\tau,t)] && (6.24)
\end{aligned}
$$

The difference $(r_j^i(t) - r_j^i(\tau))$ determines the amount of traffic arriving at node i in the interval $(t-\tau)$; the maximum of which is the sum of the tokens available at node i and the tokens received in the interval $(t-\tau)$. At every node, each destination is constrained independently by a permit bucket scheme. Following Parekh and Gallager's approximation [PG94], we assume the links to be of infinite capacity. The results for the infinite capacity case upper-bound the finite capacity case. In other words, the results of infinite capacity can be used for any finite speed link. The arrival and the service functions at each router can be translated to permit bucket parameters, which in turn depend on the maximum tolerable path delay and the link flows. Substituting for the arrivals and the number of packets serviced in terms of the permit bucket parameters from the previous section we have

$$
\begin{aligned}
Q_j^{i*}(\tau,t) &\leq [\sigma_j^i(t-\tau) + \rho_j^i(t-\tau)] - \rho_j^i(t-\tau) \\
&\quad + \sum_{l|i\in SM_j^l(t)} \frac{f_{jl}^i}{t_j^i}[\sigma_j^i(t-\tau) + \rho_j^i(t-\tau)]
\end{aligned}
$$

Because $\frac{f_{jl}^i}{t_j^i} \leq 1$ for any j and $l \in SM_j^i(t)$,

$$Q_j^{i*}(\tau, t) \leq \sigma_j^i(t-\tau) + \sum_{l \mid i \in SM_j^l(t)} [\sigma_j^l(t-\tau) + \rho_j^l(t-\tau)] \tag{6.25}$$

Making $\tau = t - \Delta t$, we can write

$$Q_j^{i*}(t) \leq \sigma_j^i(t) + \sum_{l \mid i \in SM_j^l(t)} [\sigma_j^l(t) + \rho_j^l(t)]$$

Therefore, the backlog at node i to destination j depends on the leaky bucket parameters at node i and the permit bucket parameters of all the upstream neighbors of i for which node i is in the shortest multipath set.

The delay at each node i can be computed as the weighted average path delay through all its multipath neighbors; therefore,

$$D_j^i(t) = \sum_{k \in SM_j^i(t)} \phi_{jk}^i(t) \breve{D}_{jk}^i(t) \tag{6.26}$$

The distance from i to j through neighbor k can be expressed as the sum of the distance from k to j and the link cost from i to k. The link cost is the sum of the distance and the propagation delay of that link. Therefore,

$$\begin{aligned}
\breve{D}_{jk}^i(t) &= D_{jk}^i(t) + d_{ik}(t) = D_{jk}^i(t) + [d_{jk}^i(t) + \delta_k^i(t)] \\
D_j^i(t) &= \sum_{k \in SM_j^i(t)} \phi_{jk}^i(t)[D_{jk}^i(t) + (d_k^i(t) + \delta_k^i)(t)]
\end{aligned} \tag{6.27}$$

From Eq. 6.17, $d_{ik}(t) = \delta_k^i(t)[1 + Q_j^i(t)]$, which implies that

$$D_j^i(t) = \sum_{k \in SM_j^i(t)} \phi_{jk}^i(t)[D_{jk}^i(t) + \delta_k^i(t)(1 + Q_j^i(t))] \tag{6.28}$$

Because SMC must be satisfied by every $k \in SM_j^i(t)$, $D_{jk}^i(t) < MAD_j^i(t)$. Then, if $D_j^i(\tau)$ is the maximum path delay from i to j at time τ and $Q_j^{i*}(t)$ is the maximum backlog from i to j at time t, we obtain from Eq. 6.28 that

$$D_j^{i*}(t) \quad < \quad \sum_{k \in SM_j^i(t)} [\phi_{jk}^i(t).\delta_k^i(t)(1 + Q_j^{i*})(t)]$$

$$+MAD_j^i(t) \sum_{k \in SM_j^i(t)} \phi_{jk}^i(t) \tag{6.29}$$

Let the maximum link propagation delay of all the links from i to a node in $SM_j^i(t)$ be

$$\Delta_j^i(t) = \max_{k \in SM_j^i(t)} \delta_k^i(t) \tag{6.30}$$

Therefore, the maximum path delay from i to j becomes

$$\begin{aligned} D_j^{i*}(t) \quad &< \quad \Delta_j^i(t) \sum_{k \in SM_j^i(t)} \phi_{jk}^i(t)[1 + Q_j^{i*}(t)] \\ &\quad + MAD_j^i(t) \sum_{k \in SM_j^i(t)} \phi_{jk}^i(t) \end{aligned} \tag{6.31}$$

Noticing that $Q_j^{i*}(t)$ is independent of k and substituting Eq. 6.7 in Eq. 6.31 we obtain

$$D_j^{i*}(t) < \Delta_j^i(t)[1 + Q_j^{i*}(t)] + MAD_j^i(t) \tag{6.32}$$

The above equation is an upper bound on $D_j^i(t)$ that should be expected. It states that $D_j^i(t)$ must be smaller than the sum of the product of the backlog for i at node i times the maximum link propagation delay in node i's shortest multipath, plus $MAD_j^i(t)$. The first term of Eq. 6.32 corresponds to the delay incurred by sending all backlogged packets at time t to a neighbor with the longest link propagation delay. The second term corresponds to the maximum delay incurred by any neighbor receiving the backlog packets; because any such neighbor must be on $SM_j^i(t)$, that delay can be at most equal to $MAD_j^i(t)$.

Substituting Eq. 6.25 in Eq. 6.32, we can represent the same bound in terms of permit bucket parameters as follows:

$$\begin{aligned} D_j^{i*}(t) \quad &< \quad \Delta_j^i(t)\{1 + \sigma_j^i(t) \sum_{l | i \in SM_j^l(t)} [\sigma_j^l(t) + \rho_j^l(t - \tau)]\} \\ &\quad + MAD_j^i(t) \end{aligned} \tag{6.33}$$

The bound given by Equations. 6.32 and 6.33 for router i is based on a maximum delay offered by the neighbor of i and a maximum backlog allowed at router i. This is possible because of two main features of SMRA: datagrams are accepted only if routers have enough credits to ensure their delivery, and datagrams are delivered along loop-free paths. In contrast, in traditional datagram routing architectures, any datagram presented to a router is sent towards the destination, and the paths taken by such datagrams can have loops; therefore, it is not possible to ensure a finite delay for the entry router or any relay router servicing a datagram.

6.3.2 Non-negligible Packet Size

In PGPS networks, routing nodes do not transmit packets until a packet has completely arrived. Therefore, the number of packets which will reach a downstream node is at the most equal to the number of packets serviced by its upstream neighbors. Let L_i be the maximum packet size at node i. The PGPS server does not begin servicing a packet until the last bit has arrived.

For a packet-switched network

$$A_j^i(\tau, t) = r_j^i(\tau, t) + \sum_{m \mid i \in SM_j^m(t)} S_{ji}^m(\tau, t) \tag{6.34}$$

Here, $S_{ji}^m(\tau, t)$ represents the number of packets serviced by an upstream neighbor m for which i is in the shortest multipath to j in the interval $(t - \tau)$. Let K be the number of hops in a given path from i to j; m and $m-1$ be two successive nodes. Then, for a given path,

$$\begin{aligned} \sum_{m-1} S_{jm}^{m-1}(\tau, t) &\geq A_j^m(\tau, t) - r_j^m(\tau, t) \\ &\geq \sum_{m-1} [S_{jm}^{m-1}(\tau, t) - L_{m-1} \end{aligned} \tag{6.35}$$

where, $m = 2, ..., K, \tau < t$ and L_{m-1} is the maximum length of a packet transmitted by node $m-1$. Here, the nodes m and $(m-1)$ are such that $m \in SM_j^{m-1}$.

For a PGPS system, the number of packets serviced for $t > \tau$ is given as

$$\begin{aligned} S_j^i(\tau,t) \quad &\geq \quad \min_{V\in[\tau,t]}\{[A_j^i(\tau,V) - r_j^i(\tau,V)] \\ &\quad +G_j^K(t-V)\} + K.L_i \end{aligned} \tag{6.36}$$

where V represents the last time in the interval $[\tau, t]$ at which node i begins a busy period for destination j and the function G_j^K is a convex function which indicates the amount of service given to destination j under a greedy regime.

$$\mathrm{S_j^i(0,t) \geq \min_{V\in[0,t]}\{[A_j^i(0,V) - r_j^i(0,V)] + G_j^K(t-V)\} + K.L_i} \tag{6.37}$$

With a greedy regime, the service to destination j is minimized and is delayed by an appropriate amount, which is given by the minimizing value of V, denoted by V_{min}.

$$\mathrm{S_j^i(t) \geq \{[A_j^i(V_{min}) - r_j^i(V_{min})] + G_j^K(t - V_{min})\} + K.L_i} \tag{6.38}$$

The backlog traffic for a destination j from i is the difference between the number of packets that has arrived and the number of packets serviced as in the previous case.

$$Q_j^i(0,t) = A_j^i(0,t) - S_j^i(0,t) \tag{6.39}$$

Applying similar argument as in the previous section, we have

$$Q_j^i(t) = \sigma_j^i(t) + \rho_j^i(t) + \sum_{l|SM_j^i(t)\in l} [\sigma_j^l(t) + \rho_j^l(t)] - S_j^i(t) \tag{6.40}$$

Substituting for $S_j^i(t)$ we have,

$$\begin{aligned} Q_j^i(t) \quad &= \quad \rho_j^i(t) - \rho_j^i(V_{min}) - G_j^m(t - V_{min}) + m.L_i \\ &\quad + \sum_{l|SM_j^i(t)\in l} [\sigma_j^l(t) + \rho_j^l(t)] \end{aligned} \tag{6.41}$$

Thus, the maximum backlog is given by

$$\begin{aligned} Q_j^{i*}(t) &= \rho_j^i(t) - \rho_j^i(V_{min}) + m.L_{max} - G_j^m(t - V_{min}) \\ &+ \sum_{l|SM_j^i(t) \in l} [\sigma_j^l(t) + \rho_j^l(t)] \end{aligned} \tag{6.42}$$

Having bound the worst-case backlog, we can use a similar approach as in Eq. 6.31 to obtain bounds for the maximum path delay. Since we are considering a PGPS system, the expression for maximum path delay becomes

$$D_j^{i*}(t) \leq MAD_j^i(t) + \Delta_j^i(t) \sum_{l|i \in SM_j^l(t)} [1 + Q_j^{i*}(t)] \tag{6.43}$$

Substituting for the maximum backlog from Eq. 6.42 we obtain,

$$\begin{aligned} D_j^{i*}(t) &\leq MAD_j^i(t) + \Delta_j^i(t) \sum_{l|i \in SM_j^l(t)} \{1 + \rho_j^i(t) - \rho_j^i(V_{min}) \\ &+ m.L_{max} - G_j^m(t - V_{min}) \\ &+ \sum [\sigma_j^l(t) + \rho_j^l(t)]\} \end{aligned} \tag{6.44}$$

$$\begin{aligned} D_j^{i*} &\leq MAD_j^i(t) + \Delta_j^i(t)\{1 + \rho_j^i(t) - \rho_j^i(V_{min}) \\ &+ m.L_{max} - G_j^m(t - V_{min}) \\ &+ \sum_{l|i \in SM_j^l(t)} [\sigma_j^l(t) + \rho_j^l(t)]\} \end{aligned} \tag{6.45}$$

Here again, the excess delay experienced by a packet depends on the network traffic as earlier. In addition, it also is a function of the packet size and depends on the entire path from source to a given destination node.

6.4 Two-Tier Architecture

To support end-to-end connections and thereby to guarantee QoS on top of a connectionless architecture, we model the network as a two-tier architecture. To the end user, the network

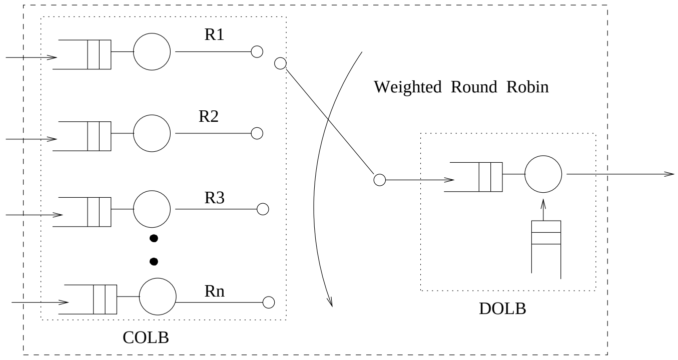

Figure 6.9: Source Model

looks like a connection oriented network through which the user can request for several levels of QoS as in any connection-oriented scheme. Within the network packets are routed on a hop-by-hop basis. Routing decisions are made independently at each hop.

6.4.1 End-to-End Model

Each source supports connection-oriented sessions. These sessions exist on top of a connectionless (destination oriented) flow. To the end user, these flows appear as connection-oriented sessions through which they can reserve resources and can have all the flexibility of a connection oriented architecture. These session based flows are mapped on to destination-based flows. Each router has the capability to differentiate one flow from another.

Figure 6.9 shows the model at a source. Connection-oriented sessions are regulated by leaky-bucket filters, which we refer to as *connection-oriented leaky buckets* (COLB). A connection oriented session generates traffic at a rate r_i, where $i = 1,...N$ is the number of sessions that can exist at any time. Each session is identified by a source-destination pair and the type of service requested by that flow.

6.4.2 Connectionless Model

Connection-oriented sessions for the same destination belonging to the same traffic class (QoS type) are grouped together and are mapped into a connectionless flow. The connectionless flows are serviced on a per-destination basis instead of per-session basis as in connection-oriented architecture. For each set of COLB sessions, a destination-oriented leaky bucket (DOLB) is maintained. This depends on the traffic class and the destination of COLB sessions. The way in which COLB flows are fed to DOLB depends on the relative rates of the associated COLB flows.

To ensure fairness among different connections, a *weighted round robin* scheduling mechanism among all COLB flows belonging to the same group is used at each of DOLB inputs. Our protocol guarantees that all packets which enter the connectionless network source will be delivered to their respective destinations, unless a resource failure prevents it. Depending on the available resources (credits) at each hop, the traffic entering the network is regulated at the COLB stage itself, before the packets actually enter the network. This ensures no packets are dropped once they enter the network. In order to regulate the traffic flow from COLBs to DOLB, a feedback mechanism is required from DOLB to COLB, so that the leaky bucket parameters of COLB can be regulated.

Let there be N connection-oriented sessions associated with each DOLB with traffic generation rates of r_1, r_2,..., r_N respectively. Each of these sessions is controlled by leaky-bucket parameters (σ_i, ρ_i), where σ is the buffer size and ρ is the traffic generation rate. (σ_S, ρ_S) are the leaky-bucket parameters for the corresponding DOLB.

Let CR_j^S be the total available credits (or tokens) at source S for destination j. The token generation at each of the COLBs is a function of the available credits. When there are N active sources the token generation at each DOLB is given by

$$\rho_i = \frac{r_i}{\sum_{k=1}^{N} r_k} CR_j^S \tag{6.46}$$

This mechanism ensures that sources are well behaved and DOLB does not accept more data than it can handle and thus can ensure guaranteed delivery of packets once the packets enter the network.

6.4.3 Node Model

Figure 6.10 shows the model of a path. Since routing decisions are made at each hop, all nodes along the path from a source to a destination can be a potential source, contributing

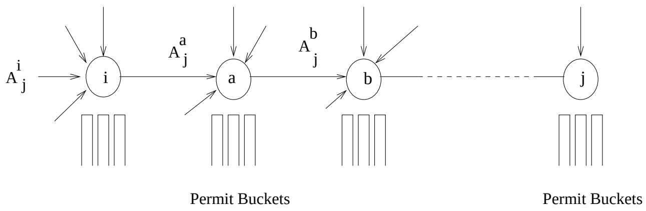

Figure 6.10: Node Model

to the traffic flow towards the destination. Therefore, the traffic at a node corresponds to the traffic arriving at a node from its upstream neighbors and the traffic originated at the node itself. Scheduling at a node is done by maintaining permit bucket filters at each node for all active destinations.

6.4.4 Flow Multiplexing

A flow is identified by a source-destination pair and a type of flow that specifies the quality of service requested. Flows having the same flow specification are grouped together and are associated with a separate set of destination-based credits. This ensures that flows with the same characteristics are guaranteed similar service.

At the source, COLB sources having the same flow specifications are multiplexed into a DOLB source. Packets from DOLB source to the destination are routed using credit-based shortest multipath routing algorithm as explained in the earlier sections. At each destination, packets are demultiplexed into respective flows and then again demultiplexed on to the COLB destinations from DOLB destinations.

The QoS required by the application can be divided into different classes and each class is serviced separately, independent of other flows. This ensures that if one service class misbehaves, it will not affect other classes or flows (flow isolation). Flow isolation is made possible because of the PGPS servers used at each node.

Figure 6.11 gives the pseudocode for credit distribution at the source. The procedure *Credit_Request* sends the explicit credit requests and the procedure *Credit_Response* demultiplexes the available credits among COLBs on receipt of the response.

Procedure Credit_Request
when a new COLB source becomes active
begin
    save state information at source
    send request (credit, seq. no)
end

Procedure Credit_Response
when a credit response is received
begin
    match the seq. no. with credit request
    distribute credits proportionally among requested sources
end

Figure 6.11: Credit Distribution at Source

6.5 Correctness of Scheduling Mechanism

In this section we prove that the scheduling policy which we have used to map connection-oriented sessions to destination-based flows is correct and live. For this purpose, we assume that all sources are well-behaved.

Lemma 6.3 *The destination-oriented leaky-bucket always demultiplexes the credits fairly among the connection-oriented sessions.*

Proof: Each DOLB is fed by a series of COLBs. Let N be the number of COLBs associated with each DOLB at any given time. When a new COLB source becomes active, DOLB sends an explicit request for credits towards destination j. Each of these explicit requests is associated with a sequence number.

When such a request is sent to the shortest multipath neighbors, the state of the source along with the sequence number is saved i.e., the COLB sources due to which an explicit request was initiated. When a credit response (analogous to an acknowledgment) for that sequence number is received, the credits are fairly distributed among sources due to which request was sent. i.e.,

$$CR_j^i = \frac{r_i}{\sum_a r_a} CR_j^S$$

where a is the set of active sources. Therefore, credit demultiplexing at DOLB is fair. This proves Lemma 6.3. $\Box$

Lemma 6.4 *The scheduling mechanism is live.*

Proof: Whenever a new source becomes active, credits are explicitly requested by that source for a specified destination through a explicit credit request message. Since each of these messages are associated with a sequence number, according to Lemma 6.3, when the source receives a credit response, credits are assigned proportionally to all requested sources. In the worst case, a credit request might have to go all the way up to the destination before it gets desired allocation. Since in steady state, we maintain loop-free paths to all destinations, the hop-count from source to a destination is finite.

Therefore, the scheduling mechanism is live. This proves Lemma 6.4. □

Theorem 8 *The scheduling mechanism is correct.*

Proof: Lemma 6.3 and 6.4 ensure that the scheduling mechanism will not have any deadlocks and all the active sources have a fair share of the available credits at all times. This proves theorem 8. □

6.6 Supporting Subnets

Having defined the basic protocol to support several QoS types in a packet-switched network, we extend it to larger networks with hierarchical topologies in order to make the protocol scalable.

6.6.1 Credit Aggregation

Destination-oriented credits are aggregated at all intermediate nodes from destination towards source. Therefore, a credit value at each hop gives the total credits available to a specified destination through all the feasible paths through its downstream nodes.

For the purpose of credit aggregation, we assume that the sources are well-behaved. Let us assume that the network is divided into clusters. Each cluster is viewed as a single entity from any node outside the cluster. All the information (distribution of credits) within the cluster is transparent to the nodes outside the cluster. The border node at each cluster is responsible for sending aggregated credit information to all the nodes/clusters outside its own cluster and then to distribute the credits fairly among the routers within the cluster.

Available credits are aggregated for each destination (cluster) at all routers (boundary nodes) along the path from a destination to a source. When a source becomes active, it sends

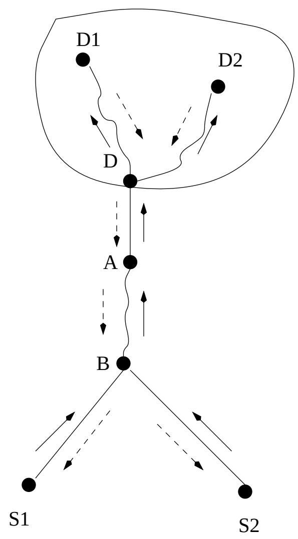

Figure 6.12: Credit Aggregation

an explicit credit request towards the desired destination. Destination in turn redistributes credits (reserving resources along the way). The number of packets entering the network is controlled at the COLB sources itself. Because of this, the sources cannot misbehave and hog the resources allocated for other sources. The COLB sources are not allowed to send data unless we have credits to do so. This ensures that the sources will not misbehave.

Because of the well-behavior of the sources, we can aggregate the available credits for destinations. This can be done without maintaining any additional information about the subnetworks. i.e., the topology of the subnetwork is transparent to the routers outside the subnetwork. Therefore, the credit-based approach is scalable.

Figure 6.12 shows an example of how the credits are aggregated in a subnet. $S1$ and $S2$ are the two sources and $D1$ and $D2$ are the two destination for which the sources are sending data. The solid line indicates the direction of the data flow and the dashed line indicates the direction of the credit flow. But, from the point of view of the two sources, the information about both the destination have been integrated into a single destination D, which is the boundary node of the destination cluster. D aggregates the credits available for the two destinations within its cluster and passes on that information towards sources $S1$ and $S2$.

Because the sources are well behaved and cannot send data to the destination unless they have credits to do so, the credits obtained are distributed between the two sources in proportion to their traffic arrival rate.

6.6.2 Fairness

Fairness can be defined as two sources of the same capacity (arrival rate), destined for the same destination and having the same QoS requirement get an equal share of the bandwidth. Fairness can be ensured in the proposed destination oriented credit mechanism among all active destinations as all destinations are served according to the traffic rate using a weighted fair queueing scheme at all nodes. The available credit is distributed among active destinations in proportions relative to their traffic arrival rate. Furthermore, the congestion control scheme is exercised hop-by-hop on a per destination basis, so that the congestion flow for one destination do not block the flows for other destinations.

At each hop, credits are allocated to flows depending on its relative input traffic rate into that node. Because the sources can not pump packets into the network unless they have credits to do so (sources are well behaved), all sources get a fair share of the available bandwidth.

6.7 Summary

In this chapter, we have proposed a new approach for combining routing and congestion control in a packet-switched network. Our protocol is based on hop-by-hop credit distribution and a loop-free multipath routing algorithm based on LPA. The new framework dynamically adapts to congestion so that the entries in the routing tables reflect the state of the network at any given time. We have demonstrated that it is possible to provide some performance bounds for the delivery of packets in such networks. A two-tier architecture to support end-to-end connections on top of a connectionless architecture is also presented to support several levels of quality of service. We have also showed that it is possible to extend this solution to a large networks. Our protocol guarantees that in the proposed two-tier architecture, all packets that enter the connectionless network will be delivered to their respective destinations unless a resource failure prevents it.

Chapter 7
Summary and Future Work

This dissertation addresses the design and analysis of routing algorithms and protocols for packet-switched networks. We identified the basic drawbacks of the existing routing techniques and also identified the requirements for new applications, which include wireless communication as well as wired applications. Different types of routing protocols are required in the network in order to support these diverse requirements. Low protocol overhead, scalability and ability to quickly adapt to the dynamic state of the network are some of the issues to be concerned about. In some cases, data traffic or best-effort traffic, does not suffice the application requirements. The main objective of this dissertation has been to take advantage of the application requirements and to propose an appropriate loop-free routing protocol.

The performance of an application can be improved by correctly designing a routing algorithm and thereby minimizing the protocol overhead. It is thus necessary to identify the application requirements for a correct design of the routing algorithm. It is also a fact that the performance of routing algorithms is greatly affected by the congestion control and the feedback mechanism of the system. For this reason, we integrate both congestion control and routing functions in one of our proposals.

We have formally verified our proposals, proving the algorithms to be correct and loop-free when applicable and have analyzed the complexity of the algorithms. We have also obtained a worst-case delay bound on the congestion oriented protocol using analytical techniques. For most of our proposals, the performance of the routing algorithms has been evaluated by means of simulations. Simulations were performed using a C-based routing simulator called *Drama*. Well known topologies such as Los-Nettos, Nsfnet and Arpanet were used in most of the simulations. To simulate our hierarchical routing algorithm, we have used the modified Doe-Esnet topology and a randomly generated topology with more then 100 nodes.

7.1 New Path-Finding Algorithms

Two new routing algorithms, PFA and LPA, which belong to the class of path-finding algorithms have been proposed. These algorithms reduce the possibility of formation of temporary routing loops. This is due to an efficient updating mechanism and the usage of implicit path information which can be extracted from the router's database maintained at each node. LPA achieves loop-freedom at every instant using a single-hop inter-neighbor coordination mechanism.

7.2 Hierarchical Routing Algorithm

The basic path-finding algorithms for flat networks have been extended to large, hierarchical networks to make the algorithms scalable. Here, the concept of maintaining predecessor information for the paths to achieve loop-free paths has been extended across the global network. This scheme ensures that the topological details within each subnet are transparent to all nodes outside the subnetwork and we achieve loop free paths across the subnetworks. This property makes the hierarchical routing algorithm scalable. Only one entry per subnetwork and the detailed information about all the nodes within the local subnet needs to be maintained at each node. This technique can be used in the Internet to increase the scalability of routing algorithms.

7.3 Wireless Routing Protocol

The requirements of a wireless network are very different from a wireline network mainly due to the limitation in the available bandwidth. WRP is a routing protocol that is suitable in a wireless environment and which takes into account the constraints of a wireless environment. This is a simple protocol which does not need any elaborate synchronization mechanism and has fast convergence properties. To limit the number of routing control messages exchanged between nodes, we suggest some optimization techniques. WRP is shown to perform better than state of the art routing protocols in wireless networks through simulations.

7.4 Congestion-Oriented Routing

Providing Quality-of-Service (QoS) guarantees in packet-switched networks places stringent demands on the underlying system. It is a well known that congestion control and routing are interrelated problems. We propose a technique to combine routing and congestion control functions and provide a multipath solution to the above problem. Worst-case delay bounds have been provided using analytical techniques. This model provides flow isolation because of the PGPS scheduling scheme used at each node. To support the quality of service required by the applications in a packet-switched environment, a two-tier architecture has been defined wherein a connection-oriented session sits on top of a packet-switched flow. A credit based protocol is used to ensure guaranteed flows within the packet-switched network and a scheduling mechanism is used to map connection-oriented sessions to packet-switched flows. The correctness of both the credit-based mechanism and the scheduling scheme have been proved analytically.

7.5 Future Work

Routing Information Protocol (RIP) [Hed88] is still the most widely used intra-domain routing protocol. Like many other protocols, RIP is also based on Distributed Bellman-Ford algorithm (DBF) for shortest path computations. To cope with the counting-to-infinity problems of DBF, the longest path in RIP is limited to 15 hops. Recently, RIP version 2 [Mal94] has been proposed, which adds more information to RIP and reports the next hop information in the routing updates. However, this still does not prevent the counting-to-infinity problem. However, the format of RIP version 2 permits the specification of limited path information. Our work can be extended to propose a new Internet routing protocol, which overcomes the drawbacks of RIP using the same packet format at RIP version 2.

The integration of congestion control and routing is an important result since the routing tables entries can be used to define long term connections along the non-congested paths. This proposal uses a multipath routing algorithm. An immediate extension of this work will be to study the behavior of multicast connections and combining it with congestion control issues to define long-term multicast sessions.

Another extension is to study the mobility issues in conjunction with wireless routing. Issues such as hierarchical clustering and hand-offs needs to be addressed. In addition to this, we can introduce another level of complexity in the study by providing guaranteed flows based on application requirements (issue of QoS).

Finally, the scheduling mechanism that is used to map connection-oriented sessions to packet-switched flows can be a topic of further research.

Some work is also being done in securing distance-vector routing protocols. [SMGLA96] gives the details of this protocol.

Bibliography

[Ana96] Arul Ananthanarayanan. Implementation of path-finding routing algorithms for unix. Master's thesis, San Diego State University, 1996.

[Bea89] D. Beyer and et. al. Packet radio network research, development and application. In *Proceedings SHAPE conference on Packet Radio*, Amesterdam, 1989.

[Bel57] R.E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, N.J., 1957.

[Ber82] D. Bertsekas. Dynamic behavior of shortest path routing algorithms. *IEEE Trans. Automatic Control*, AC-27:60–74, 1982.

[Bey90] D. Beyer. Accomplishments of the DARPA SURAN program. In *MILCOM*, pages 855–862, Monterey, California, Dec. 1990. IEEE.

[BG92] D. Bertsekas and R. Gallager. *Data Networks.* Prentice-Hall Inc, 2nd edition, 1992.

[BJ83] A.E. Baratz and J.M. Jaffe. Establishing virtual circuits in large computer netwroks. In *INFOCOM*, pages 311–318, San Diego, CA, April 1983. IEEE.

[CE95] M. Scott Corson and Anthony Ephremides. A distributed routing algorithm for mobile wireless networks. *ACM J. Wireless Networks*, pages 61–81, Jan. 1995.

[Ceg75] T. Cegrell. A routing procedure for the TIDAS message switching network. *IEEE Trans. Commun.*, 23(6):575–585, 1975.

[CRKGLA89] C. Cheng, R. Reley, S.P.R. Kumar, and J.J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. In *Computer Communications Review*, volume 19, pages 224–236. ACM, 1989.

[Cru91] R.L. Cruz. A calculus for network delay, part II: Network analysis. *IEEE Trans. Information Theory*, 37:132–141, 1991.

[DH96] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification. RFC 1883, Jan. 1996.

[DS80] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. In *Information Processing Letters*, volume 11. 1980.

[Far93] Dino Farinacci. Introduction to enhanced IGRP. Cisco Systems, June 1993.

[FJ94] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Trans. Networking*, 2(2):122–136, April 1994.

[GLA] J.J. Garcia-Luna-Aceves. A minimum-hop routing algorithm based on distributed information. *Computer Networks and ISDN Systems*, 16:367–382.

[GLA86] J.J. Garcia-Luna-Aceves. A fail-safe routing algorithm for multihop packet radio networks. In *INFOCOM*, Maiami, Florida, April 1986. IEEE.

[GLA92a] J.J. Garcia-Luna-Aceves. Distributed routing with labeled distances. In *INFOCOM*, pages 633–643. IEEE, 1992.

[GLA92b] J.J. Garcia-Luna-Aceves. Libra: A distributed routing algorithm for large internets. In *GLOBECOM*, volume 3, pages 1465–1471. IEEE, Dec. 1992.

[GLAM95] J.J. Garcia-Luna-Aceves and Shree Murthy. A loop-free path-finding algorithm: Specification, verification and complexity. In *INFOCOM*, Boston, April 1995. IEEE.

[GLAM96] J.J. Garcia-Luna-Aceves and Shree Murthy. A path-finding algorithm for loop-free routing. *IEEE/ACM Trans. Networking*, 1996. to appear in.

[GLAS85] J.J. Garcia-Luna-Aceves and N. Shacham. Analysis of routing strategies for packet radio networks. In *INFOCOM*, pages 292–302, Washington D.C., March 1985. IEEE.

[GLAZ94] J.J. Garcia-Luna-Aceves and William T. Zaumen. Area-based loop-free internet routing. In *INFOCOM*, volume 3, pages 1000–1008, Toronto, Canada, April 1994. IEEE.

[Hag83] J. Hagouel. Issues in routing for large dynamic networks. Technical Report 9942 (No. 44055, IBM Research Report, April 1983.

[Hed] Charles Hedrick. An introduction to IGRP. Rutgers University.

[Hed88] C. Hedrick. Routing information protocol. RFC 1058, June 1988.

[HS82] R. Hinden and A. Sheltzer. DARPA internet gateway. RFC 823, Sept. 1982.

[Hui95] Christian Huitema. *Routing in the Internet*. Prentice Hall, 1995.

[Hum91] P.A. Humblet. Another adaptive shortest-path algorithm. *IEEE Trans. Commun.*, 39(6):995–1003, June 1991.

[Jaf88] J.M. Jaffe. Hierarchical clustering with topology databases. *Computer Networks and ISDN Systems*, 15:329–339, 1988.

[JM82] J.M. Jaffe and F.M. Moss. A responsive routing algorithm for computer networks. *IEEE Trans. Commun.*, 30:1758–1762, July 1982.

[Kam76] F. Kamoun. *Design Considerations for Large Computer Communications Network*. PhD thesis, University of California, Santa Cruz, 1976. UCLA-ENG-7642, Computer Science Dept.

[KTA94] H.T Kung, Blackwell. T, and Chapman. A. Credit-based flow control for ATM networks: credit update protocol, adaptive credit allocation, and statistical multiplexing. In *ACM SIGCOMM*, volume 24 of *CCR*, pages 101–14, London, UK, Aug/Sept 1994. ACM.

[Lau86] G.S. Lauer. Hierarchical routing design for suran. In *ICC*, pages 4.2.1–4.2.10, Toronto, Canada, June 1986. IEEE.

[LNT87] B.M. Leiner, D.L. Nielson, and F.A. Tobagi. Issues in packet-radio network design. volume 75 of *Proc. of IEEE*, pages 6–20, Jan. 1987.

[Mal94] G. Malkin. RIP version 2: Carrying additional information. RFC 1723, Nov. 1994.

[McQ74] J. McQuillan. Adaptive routing algorithms for distributed computer networks. Technical report, Bolt Beranek and Newman, 1974.

[MGLA94] Shree Murthy and J.J. Garcia-Luna-Aceves. A more efficient path-finding algorithm. In *28th Asilomar Conference*, pages 229–233, Pacific Groove, California, Nov. 1994.

[MGLA95] Shree Murthy and J.J. Garcia-Luna-Aceves. A routing protocol for packet-radio networks. In *Mobile Computing and Networking Conference*, Berkeley, CA, Nov. 1995. ACM.

[MGLA96] Shree Murthy and J.J. Garcia-Luna-Aceves. A routing protocol for wireless networks. *Mobile Networks and Applications*, 1(2), 1996. Special Issue on Routing in Mobile Communications Network.

[Mil83a] D. Mills. DCN local network protocols. RFC 891, Dec. 1983.

[Mil83b] D. Mills. Exterior gateway protocol. RFC 904, Dec. 1983.

[Moy94] J. Moy. OSPF version 2. RFC 1583, March 1994.

[MRR78] J.M. McQuillan, I. Richer, and E.C. Rosen. Arpanet routing algorithm improvements. Technical Report 3803, BBN, April 1978.

[MRR80] J. McQuillan, I. Richer, and E. Rosen. The new routing algorithm for the Arpanet. *IEEE Trans. on Commun.*, COM-28(5):711–19, May 1980.

[MS79] P.M. Meerlin and A. Segall. A failsafe distributed routing algorithm. *IEEE Trans. Commun.*, 27:1280–1288, Sept. 1979.

[Mur94] Shree Murthy. Design and analysis of distributed routing algorithms. Master's thesis, University of California, Santa Cruz, June 1994.

[MW77] J. McQuillan and D.C. Walden. The ARPANET design decisions. *Computer Networks*, 1, Aug. 1977.

[Ora90] D. Oran. OSI IS-IS intra-domain routing protocol. RFC 1247, Feb. 1990.

[OSV94] C. Ozveren, R. Simcoe, and G. Varghese. Reliable and efficient hop-by-hop flow control. In *SIGCOMM*, pages 89–110, London, Aug/Sept. 1994.

[PB94] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *SIGCOMM*, pages 234–244. ACM, 1994.

[Per91] Radia Perlman. A comparison between two routing protocols: OSPF and IS-IS. In *IEEE Network*, volume 5, pages 18–24, Sept. 1991.

[PG93] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Trans. Networking*, 1(3):344–357, June 1993.

[PG94] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. *IEEE/ACM Trans. Networking*, 2(2):137–150, April 1994.

[RF91] B. Rajagopalan and M. Faiman. A responsive distributed shortest-path routing algorithm within autonomous systems. *Internetworking: Research and Experience*, 2(1):51–69, March 1991.

[RL94] Y. Rekhter and T. Li. Border gateway protocol 4 (BGP-4). RFC, Jan. 1994.

[RT83] C.V. Ramamoorthy and W. Tsai. An adaptive hierarchical routing algorithm. In *COMPSAC*, pages 93–104. IEEE, Nov. 1983.

[Sch86] M. Schwartz. *Telecommunications Networks: Protocols, Modeling and Analysis*. Addison-Wesley Publishing Co.,, Menlo Park, California, 1986. Chapter 6.

[SK86] J. Seeger and A. Khanna. Reducing routing overhead in a growing DDN. In *MILCOM*, volume 1, pages 15.3.1–15.3.13. IEEE, Oct. 1986.

[SMGLA96] Brad Smith, Shree Murthy, and J.J. Garcia-Luna-Aceves. Securing distance-vector routing protocols. UCSC Technical Report, 1996.

[Tan91] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, second edition, 1991.

[Wax88] B.M. Waxman. Routing of multipoint connections. *J. Selected Areas in Commun.*, 6(9):1617–1622, 1988.

[Zau91] W.T. Zaumen. Simulations in drama. Network Information System Center, SRI International, Jan. 1991.

[ZDE+93] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. Rsvp: A new resource reservation protocol. In *Network Magazine*, volume 7, pages 8–18. IEEE, Sept. 1993.

[ZGLA92] W.T. Zaumen and J.J. Garcia-Luna-Aceves. Dynamics of link-state and loop-free distance-vector routing algorithms. In *Internetworking: Research and Experience*, volume 3, pages 161–188. 1992.