SELF-STABILIZING INTER-DOMAIN POLICY ROUTING

by

Venkatakrishnan Trichy Ramasubramanian

Bachelor of Science
Bharathiar University, India
2001

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2004
The Thesis prepared by

VENKATAKRISHNAN TRICHY RAMASUBRAMANIAN

Entitled

SELF-STABILIZING INTER DOMAIN POLICY ROUTING

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Sebastian Tixeuil
Examination Committee Co-Chair

Ajay Kumar Dutta
Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative
ABSTRACT

Self-stabilizing Inter-Domain Policy Routing

by

Venkatakrishnan Trichy Ramasubramanian

Dr. Ajoy K. Datta, Examination Committee Chair
School of Computer Science
University of Nevada, Las Vegas

Dr. Sébastien Tixeuil, Examination Committee Co-Chair
Professor of Computer Science
Université de Paris-Sud, France

The global Internet is composed of many autonomous systems. An autonomous system is a collection of hosts, gateways, networks and links managed by a single administrative authority. Routing is the task of determining the best neighbor to which a data message should be forwarded towards a particular destination. Policy routing refers to any form of routing that is influenced by factors other than merely picking the shortest path. The quality of service and policy compatibility can all be expressed as a set of desired path attributes. Policy routing searches for a path that meets all the requested attributes. Inter Domain Policy Routing (IDPR) is used to construct and maintain routes between different autonomous systems. These routes provide user traffic with the services requested within the constraints stipulated for the autonomous systems transited.

As the complexity of the networked systems increases, the likelihood of experiencing unanticipated faults grows. Self-stabilization is the most general technique to design fault tolerant systems. This paradigm was introduced by Dijkstra in 1974. A self-stabilizing
system guarantees that starting from an arbitrary state, the system converges to a legal state in a finite number of steps and remains in a legal state until another fault occurs. Such a system after any unexpected perturbation eventually recovers without any outside intervention.

The goal of this thesis research is to design a self-stabilizing Inter Domain Policy Routing Algorithm in order to make the policy routing procedure resistant to failures. We propose two algorithms. Our first algorithm sets up a path from a source to a destination, where the source and destination belong to different autonomous systems. It uses the Path Control Protocol (PCP). This algorithm can handle path failures, but cannot cope with message losses. Our second solution takes care of this problem by using the Control Message Transport Protocol (CMTP). Formal proof of correctness for both solutions will be given.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>vii</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1. Our Contributions</td>
<td>2</td>
</tr>
<tr>
<td>2. Thesis Outline</td>
<td>2</td>
</tr>
<tr>
<td>CHAPTER 2 BACKGROUND CONCEPTS</td>
<td>3</td>
</tr>
<tr>
<td>3. Routing in the Internet</td>
<td>3</td>
</tr>
<tr>
<td>4. Autonomous System</td>
<td>5</td>
</tr>
<tr>
<td>5. Policy Routing</td>
<td>5</td>
</tr>
<tr>
<td>6. Policy Route Generation</td>
<td>5</td>
</tr>
<tr>
<td>7. Message Forwarding</td>
<td>6</td>
</tr>
<tr>
<td>8. Border Gateway Protocol</td>
<td>6</td>
</tr>
<tr>
<td>9. Self-stabilization</td>
<td>7</td>
</tr>
<tr>
<td>CHAPTER 3 INTER-DOMAIN POLICY ROUTING</td>
<td>10</td>
</tr>
<tr>
<td>10. Policies</td>
<td>10</td>
</tr>
<tr>
<td>11. Routing and Message Forwarding</td>
<td>11</td>
</tr>
<tr>
<td>12. Functions</td>
<td>11</td>
</tr>
<tr>
<td>13. Elements</td>
<td>12</td>
</tr>
<tr>
<td>15. Approach</td>
<td>15</td>
</tr>
<tr>
<td>CHAPTER 4 SELF-STABILIZING IDPR</td>
<td>17</td>
</tr>
<tr>
<td>17. Motivation</td>
<td>17</td>
</tr>
<tr>
<td>18. Related Work</td>
<td>18</td>
</tr>
<tr>
<td>19. Model and Preliminaries</td>
<td>19</td>
</tr>
<tr>
<td>20. Model</td>
<td>19</td>
</tr>
<tr>
<td>21. Problem Specification</td>
<td>21</td>
</tr>
<tr>
<td>22. Inter Domain Policy Routing Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>23. Data Structures</td>
<td>22</td>
</tr>
<tr>
<td>24. Informal Description</td>
<td>23</td>
</tr>
<tr>
<td>25. Proof of Correctness</td>
<td>23</td>
</tr>
<tr>
<td>26. Reliable Delivery</td>
<td>23</td>
</tr>
<tr>
<td>27. Convergence</td>
<td>23</td>
</tr>
<tr>
<td>28. Pseudo Code for the Algorithms</td>
<td>23</td>
</tr>
<tr>
<td>29. Data Structures (PCP)</td>
<td>23</td>
</tr>
<tr>
<td>30. Abstract Version (PCP)</td>
<td>42</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 5 CONCLUSION AND FUTURE WORK

BIBLIOGRAPHY

VITA
ACKNOWLEDGMENTS

I am grateful to my thesis advisor, Dr. Ajoy K. Datta, for chairing my committee and advising this work. I am indebted to him for making my thesis possible and for motivating this research. I owe a lot to the co-chair of the examination committee, Dr. Sébastien Tixeuil at Université de Paris-Sud, France, for his initial ideas and guidance throughout my work. I would like to give my special thanks to Dr. John Minor, Dr. Wolfgang Bein, and Dr. Venkatesan Muthukumar for their participation in my thesis committee. I would also like to thank the School of Computer Science at the University of Nevada Las Vegas for offering me financial support throughout my graduate studies.

This thesis is dedicated to my parents, brother, and sister-in-law for their support and prayers. I would like to thank my friend Priya for her encouragement and help during this research. I am grateful to Isai, Supriya, and Preethi for sharing their knowledge on this topic. I would like to thank Chaitanya, Kiran, Rajesh, Sachin, and Senthil for their support and friendship.
CHAPTER 1

INTRODUCTION

The Internet has evolved from a single homogeneous environment into the infrastructure that consists of a multivendor collection of diverse network technologies. It is a massive internetwork of router-connected networks, consisting of thousands of autonomous systems that are managed by different authorities. These networks share traffic and exchange routing information.

The commercialization of Internet has initiated the need of policy based routing. In today's high performance internetworks, organizations need the freedom to implement packet forwarding and routing according to their own defined policies in a way that goes beyond traditional routing protocol concerns. In cases where administrative issues dictate that traffic be routed through specific paths, policy-based routing can provide the solution. By using policy-based routing, customers can implement policies that selectively cause packets to take different paths [1]. Policy routing also provides a mechanism to mark packets so that certain types of traffic receive preferential service. Inter-Domain Policy Routing (IDPR) is used to construct and maintain policy routes among autonomous systems. It ensures that the traffic is forwarded along routes that offer the requested services without violating the restrictions of intermediate autonomous systems.

Various types of faults can occur in inter domain routing. Messages may get lost or
corrupted. The memory variables can get corrupted. Thus, routing among autonomous systems must be fault-tolerant. The paradigm of self-stabilization introduced by Dijkstra [18] is the most unified approach to design fault-tolerant systems.

1.1 Our Contributions

In this thesis, we will study the application of self-stabilization in designing the Inter Domain Policy Routing algorithm. We solve the problem of setting up a path from a source to a destination, where source and destination belong to different autonomous systems. The path satisfies the requested services while respecting the transit policies of the intermediate autonomous systems.

Our solution is self-stabilizing. It handles many types of faults, including message losses and variable corruptions. We send data after conforming the successful establishment of the path. Our solution can also handle path failures. We present two algorithms. The first algorithm sets up the path using the Path Control Protocol (PCP). The second algorithm deals with message losses using the Control Message Transport Protocol (CMTP).

1.2 Thesis Outline

In chapter 2, we give an overview of some of the areas involved in this research, such as routing, Internet, and self-stabilization. In Chapter 3, the functions, elements, policies of IDPR are briefly explained. We present 2 algorithms for self-stabilizing Inter-Domain Policy Routing, followed by its proof of correctness in Chapter 4.
CHAPTER 2

BACKGROUND CONCEPTS

In this chapter, we explain some terms used in routing, Internet, and self-stabilization. In Section 2.1, we discuss routing in the Internet, and in Section 2.2, we define the Autonomous System. A brief discussion of policy routing is included in Section 2.3. A brief overview of Border Gateway Protocol (BGP) is presented in Section 2.4. In Section 2.5, we give a short introduction to self-stabilization.

2.1 Routing in the Internet

The Internet is not "one network", there is no such thing as one huge international company that would provide connections to users in various continents. Instead, the Internet is a loose interconnection of networks belonging to many owners. One usually distinguishes three levels of networks: organizational, regional, and transit. The companies and institutions attached to the Internet generally manage an internal network. Its complexity can vary with the size of the organization. Most organization's networks are connected to the Internet through a "regional" provider which manages a set of links covering a state, a region, or a small country. These regional networks provide connectivity to their customers. They also render a number of related services such as helping users to manage their networks, getting Internet addresses, and providing mail-boxes for isolated users. Being connected to
other Internet users in the same city, even in the same state, is not quite sufficient. The purpose of Internet is worldwide connectedness. This connectivity may be provided by a "transit" provider [29].

Every packet has a source and a destination, and routing is the mechanism which determines the path a packet should take in order to reach the specified destination. Routing on the Internet can be classified into two areas: local routing and wide-area routing [23]. Local routing transports a packet to a host within a particular network once it has reached that network. Wide-area routing deals with transporting a packet between networks, i.e., across the Internet itself.

Within any host, there will be a routing table that the host uses to determine the physical interface address to use for the outgoing packets. If a computer receives a packet on any interface, there are two possibilities. The packet is intended for the computer. Then the packet will be passed to the relevant application. Otherwise, the packet is addressed to some other computer. We re-transmit on one of the available interfaces towards that computer.

Internet routing instability, or the rapid fluctuation of network reachability information, is an important problem currently facing the Internet engineering community. High levels of network instability can lead to packet loss, increased network latency and time to converge. At the extreme, high levels of routing instability have led to the loss of internal connectivity in wide-area, national networks [12].

Administrative policies, performance requirements, load balancing, and scalability are becoming increasingly significant factors in Internet routing. Intelligent path selection based on multiple constraints or packet content takes these factors into consideration. Constraint-
based routing (CBR) denotes a class of routing algorithms that base path selection decisions on a set of requirements or constraints in addition to the destination. These constraints may be imposed by administrative policies or by Quality of Service (QoS) requirements. Constraints imposed by policies are referred to as policy constraints, and the associated routing is referred to as policy routing (or policy-based routing). Constraints imposed by QoS requirements, such as bandwidth, delay, or loss are referred to as QoS constraints, and the associated routing is referred to as QoS routing [40].

2.2 Autonomous System

An Autonomous System (AS) is a collection of hosts, gateways, networks, and links managed by a single administrative authority. The domain administrator defines service restrictions for transit traffic, and service requirements for locally-generated traffic. It also selects the addressing schemes and routing procedures that apply within the domain. The minimal AS is composed of exactly one router directly connecting one local network to the Internet, but there is no theoretical limit to the size of an AS. We will use the terms "autonomous systems" and "administrative domains" interchangeably in this report.

2.3 Policy Routing

In its broadest sense, policy routing refers to any form of routing that is influenced by factors other than merely picking the shortest path [29]. Policy routing is used to accommodate "acceptable-use policies", to select providers and to find paths that provide a particular quality of service. For example, commercial users do not want to send packets through academic backbones. They have to route packets through a network that accepts commercial traffic. They can specify the list of providers to be used so that it meets the
throughput or delay requirements. The quality of service considerations such as finding a path that would be adequate for high-bandwidth can also be included. The quality of service and policy compatibility can all be expressed as a set of desired path attributes.

2.3.1 Policy Route Generation

**Distance Vector Approach:** This type of routing protocol requires that each router inform its neighbors of its routing table. For each network path, the receiving routers pick the neighbor advertising the lowest cost, then add this entry into its routing table for re-advertisement [2]. Distance vector route generation distributes the computation of a single route among multiple routing entities along the route. Hence, distance vector route generation is potentially susceptible to routing loop formation and slow adaptation to changes in an internetwork.

**Link State Approach:** This type of routing protocol requires each router to maintain at least a partial map of the network. When a network link changes its state (up to down, or vice versa), a notification, called a *link state advertisement* (LSA) is flooded throughout the network. All the routers note the change, and recompute their routes accordingly. This method is more reliable, easier to debug, and less bandwidth-intensive than distance vector [2]. Link state route generation permits concentration of the computation of a single route within a single routing entity at the source of the route.

2.3.2 Message Forwarding

**Hop-by-Hop Approach:** In hop-by-hop message forwarding, each routing entity makes an independent forwarding decision based on a message's source, destination, and the entity's forwarding information database.
Source Specified Approach: In source specified message forwarding, the source domain dictates the data message forwarding decisions to the routing entities in each intermediate domain. Those entities then forward data messages according to the source specification.

2.4 Border Gateway Protocol

The Border Gateway Protocol (BGP) is an inter-autonomous system routing protocol [33]. BGP exchanges network reachability information with other BGP speakers. The information for a network includes the complete list of autonomous systems that the traffic must transit to reach that network. This information can then be used to ensure loop-free paths. This information is sufficient to construct a graph of AS connectivity from which routing loops may be pruned and some policy decisions at the AS level may be enforced.

BGP uses path vector routing and hop-by-hop message forwarding. BGP (Border Gateway Protocol) is a protocol for exchanging routing information between gateway hosts (each with its own router) in a network of autonomous systems. BGP is often the protocol used between gateway hosts on the Internet. The routing table contains a list of known routers, the addresses they can reach, and a cost metric associated with the path to each router so that the best available route is chosen.

To characterize the set of policy decisions that can be enforced using BGP, one must focus on the rule that an autonomous system advertises to its neighboring autonomous only those routes that it itself uses. This rule reflects the hop-by-hop routing paradigm generally used throughout the current Internet. Note that some policies cannot be supported by the hop-by-hop routing paradigm, and thus require techniques such as source routing to enforce. For example, BGP does not enable one autonomous system to send traffic to a
neighboring autonomous system intending that traffic take a different route from that taken by the traffic originating in the neighboring autonomous system. On the other hand, BGP can support any policy conforming to the hop-by-hop routing paradigm [33].

2.5 Self-stabilization

The notion of self-stabilization was introduced by Dijkstra in 1973. He defined a system as self-stabilizing when “regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps” [17]. Lamport later appreciated and explained it.

A self-stabilizing system $S$ guarantees that, starting from an arbitrary global state, it reaches a legal global state within a finite number of state transitions, and remains in a legal state unless a change occurs. In a non-self-stabilizing system, the system designer needs to enumerate the accepted kinds of faults, such as node/link failures, and he must add special mechanisms for recovery. Generally, not all types of faults are taken in consideration, and an obscure error such as a memory corruption can provoke a general reset of the entire system. Ideally, a system should continue its work by correctly restoring the system state whenever a fault occurs [6, 25].

Self-stabilization was defined in terms of closure and convergence in [5, 6]. Closure refers to the property which requires that during all system executions, the system stays within some set of legal or desirable set of states unless a fault occurs. Convergence requires the system to reach a legal state from any arbitrary (possibly illegal) state in finite steps. A system is self-stabilizing if it satisfies both closure and convergence properties.

In [5, 6], a comprehensive study of different types of faults (such as crash, stuck-at, fail-stop, omission, timing, performance and Byzantine) and how they are accommodated...
in their definition of stabilization (in terms of closure and convergence) was included. The first formal definition of fault-tolerance was given in [6]. The results like [5, 6] and some others (refer [28]) in subsequent years establish the fact that self-stabilization is the most unified strategy of achieving fault-tolerance in distributed systems.

Numerous models have been considered in the literature. There exist several dimensions of the model, such as execution model (shared registers and message passing), fairness (weakly fair, strongly fair, and unfair), granularity of the atomic step (composite vs. read/write atomicity), and types of daemons (central and distributed). Stabilization time complexity and space complexity have been two important factors in this topic. Stabilizing a program is quite challenging. Two techniques have been commonly used in the literature: convergence stair [26] and variant function [30] methods.

Many general methods of designing self-stabilizing programs have been proposed. We mention some of them here without any description: diffusing computation [7], silent stabilization [21], local stabilizer [4], local checking and local correction [8, 38], distributed program checking [9], counter flushing [39], window washing [14], self-containment [24], snap-stabilization [15], super-stabilization [22], power supply [3], and transient fault detector [10]. A brief survey of self-* systems is given in [41].

Self-stabilization has been extensively used in the area of network protocols. Numerous papers have been written on protocols like routing (including cut-through, wormhole), alternating bit, sliding window, session control, congestion control, connection management, high-speed networks, sensor networks, and max-flow computation. Refer to [20, 28] for the pointers. Many of these protocols also consider message losses and duplications, and node/link failures.
CHAPTER 3

INTER-DOMAIN POLICY ROUTING

Inter Domain routing refers to routing among Autonomous Systems. The objective of IDPR is to construct and maintain routes between a source and a destination autonomous system. The route should provide user traffic with the services requested within the constraints stipulated for the autonomous systems transited.

In this chapter, we discuss some concepts associated with Inter-Domain Policy Routing (IDPR). In section 3.1, we describe the type of policies. Section 3.2 briefly explains the routing and message forwarding techniques used. A list of the functions performed by IDPR is presented in Section 3.3. In Section 3.4, we give a short introduction to the elements in IDPR architecture. The protocols and procedures present in IDPR are given in Section 3.5. Finally in Section 3.6, we describe how IDPR approaches policy routing in the Internet.

3.1 Policies

Transit Policy: With policy routing, each autonomous system administrator sets transit policies that dictate how and by whom the resources within its autonomous system should be used. Transit policies are usually public, and they specify offered services.

Source Policy: Each autonomous system administrator also sets source policies for traffic
originating from its autonomous system. Source policies are usually private, and they specify requested services.

3.2 Routing and Message Forwarding

**Source Specified Message Forwarding:** With source specified message forwarding, the source autonomous system dictates the data message forwarding decisions to the routing entities in each intermediate autonomous system, which then forward data messages according to the source specification. Thus, the source autonomous system ensures that any data message originating from it follows its selected routes. For source specified message forwarding, each data message must carry either an entire source specified route or a path identifier.

**Link State Routing:** In link state routing, all nodes have a copy of the network map, which is regularly updated. With link state routing information distribution, all recipients of a autonomous system's link state message gain knowledge of that autonomous system's transit policies, and hence service restrictions. Thus, an autonomous system has complete control over service restrictions and restricting distribution of its routing information.

3.3 Functions

Inter-domain policy routing is comprised of the following functions:

1. Collecting and distributing routing information including autonomous system transit policies and inter-domain connectivity.

2. Generating and selecting policy routes based on the routing information distributed and on the source policies configured or requested.
3. Setting up paths across the internet using the policy routes generated.

4. Forwarding messages across and between autonomous systems along the established paths.

5. Maintaining databases of routing information, inter-domain policy routes, and forwarding and configuration information.

3.4 Elements

From the perspective of IDPR, the Internet consists of autonomous systems connected by virtual gateways which are in turn connected by intra-domain routes supporting the transit policies configured by the autonomous system administrators.

**Virtual Gateways**: They are the only connecting points recognized by IDPR between adjacent autonomous systems. Each virtual gateway is actually a collection of directly connected policy gateways in two adjacent autonomous systems whose existence has been sanctioned by the administrators of both autonomous systems.

**Policy Gateways**: The physical gateways within a virtual gateway are policy gateways. Each policy gateway forwards transit traffic according to the service restrictions stipulated by its autonomous system's transit policies applicable to its virtual gateway. Within a virtual gateway, two policy gateways are *peers* if they are in the same autonomous system, and are *adjacent* if they are in different autonomous systems.

**Route Server**: Each route server is responsible for its database of routing information, including autonomous system connectivity and transit policy information, and the database of policy routes. Each route server generates policy routes on behalf of its autonomous system. It may be part of the functionality of the policy gateway.
Path Agents: They act on behalf of hosts to select policy routes, to set up and manage paths, and to maintain forwarding information databases. It may reside in the policy gateway.

Mapping server: It is responsible for the database of mappings that resolve Internet names and addresses to autonomous systems. The functions of the mapping server can be integrated into an existing name service such as the DNS.

Configuration server: The database of configured information that applies to policy gateways, path agents, and route servers in the given autonomous system is maintained by a configuration server. The functions of the configuration server can be integrated into the autonomous system's existing network management system.

3.5 Protocols and Procedures

Virtual Gateway Protocol: Every policy gateway within an autonomous system participates in gathering information about connectivity within and between virtual gateways of which it is a member, and in distributing this information to other virtual gateways in its autonomous system. These functions are referred to as the Virtual Gateway Protocol (VGP). Virtual gateway connectivity information, distributed to policy gateways within a single autonomous system, aids those policy gateways in selecting routes across and between virtual gateways connecting their autonomous system to adjacent autonomous systems.

Flooding Protocol: An autonomous system representative policy gateway uses unrestricted flooding among all autonomous systems to distribute its autonomous system's IDPR routing information messages to route servers in an internetwork. There are two kinds
of IDPR routing information messages issued by each autonomous system representative: CONFIGURATION and DYNAMIC messages. Each CONFIGURATION message contains the transit policy information configured by the autonomous system administrator, including for each transit policy, its identifier, specification, and the sets of virtual gateways configured as mutually reachable via intra-domain routes supporting the given transit policy. Each DYNAMIC message contains information about current virtual gateway connectivity to adjacent autonomous systems, and the sets of virtual gateways currently mutually reachable via intra-domain routes supporting the configured transit policies.

**Route Server Query Protocol**: Each route server is responsible for maintaining both the routing information database and the route database, and for responding to database information requests from policy gateways and other route servers. These requests and their responses are the messages exchanged via the Route Server Query Protocol (RSQP). Policy gateways and route servers normally invoke RSQP to replace absent, outdated, or corrupted information in their own routing information or route databases.

**Path Control Protocol**: Two entities in different autonomous systems may exchange IDPR data messages, only if there exists an IDPR path set up between the two autonomous systems. Path setup requires cooperation among path agents and intermediate policy gateways. Path agents locate policy routes, initiate the Path Control Protocol (PCP), and manage existing paths between autonomous systems. Intermediate policy gateways verify that a given policy route is consistent with their autonomous systems’ transit policies, establish the forwarding information, and forward messages
along existing paths. Messages exchanged by the path control protocol are classified into requests: SETUP, TEARDOWN, REPAIR; and responses: ACCEPT, REFUSE, and ERROR.

**Route Generation Procedure:** Route generation is the most computationally complex part of IDPR, due to the number of autonomous systems, and the number and heterogeneity of policies that it must accommodate. Route servers must generate policy routes that satisfy the requested services of the source autonomous systems, and respect the offered services of the transit autonomous systems.

**Data Message Forwarding Procedure:** Once a path for a policy route is set up, its physical realization is a set of consecutive policy gateways, with policy gateways or route servers forming the endpoints. Two successive entities in this set belong to either the same autonomous system or the same virtual gateway. A policy gateway or route server may, at any time, recover the resources dedicated to a path that goes through it by tearing down that path. For example, a policy gateway may decide to tear down a path that has not been used for some period of time.

### 3.6 Approach

The task in the design of IDPR consists of the following: maintenance of a “network map”, the computation of policy routes from this map, and the setup of paths through the interconnection of autonomous systems.

1. It would be unrealistic to represent the whole Internet in a single link state database. The autonomous systems in the Internet provide aggregates for a high-level map, where the nodes would not be the routers but the autonomous systems themselves, and the links
would be those linking the autonomous systems.

2. IDPR routes are computed on demand. A route request lists the source, the destination, the requested quality-of-service, and monetary constraints. The source and transit policies are taken into account.

3. The path setup is the establishment of a virtual circuit. The path control messages are used to set up the path. The initial setup request progresses hop-by-hop from a policy gateway to another policy gateway. Each intermediate hop checks if the request is compatible with the autonomous system's transit policies, and may refuse the path establishment if this is not the case. The path is assigned a path identifier. The path will be established when the setup message reaches the target. An acceptance message will travel towards the source, after which data can be sent by the source.
CHAPTER 4

SELF-STABILIZING IDPR

In this chapter, we present a self-stabilizing IDPR algorithm. First, in Section 4.2, we discuss some of the fault-tolerant routing techniques used in Inter-Domain Routing. In Section 4.3 we describe the model, network topology, and the specification of the problem. The IDPR Algorithm and its description are given in Section 4.4. In Section 4.5, we present the proof of correctness. The pseudo code of the algorithms are presented in Section 4.6.

4.1 Motivation

As data communication technologies evolve and user populations grow, the demand for internetworking increases. Internetworks usually proliferate through interconnection of autonomous and heterogeneous networks administered by separate authorities. Interconnection of administrative domains can broaden the range of services available in an internetwork. Hence, traffic with special service requirements is more likely to receive the service requested. However, administrators of domains offering special transit services are more likely to establish stringent access restrictions in order to maintain control over the use of their domains' resources. An internetwork composed of many domains with diverse service requirements and restrictions requires policy routing to transport traffic between source and destination [34]. Policy routing constitutes route generation and message for-
warding procedures for producing and using routes that simultaneously satisfy user service requirements and respect transit domain service restrictions. In the large and heterogeneous Internet, the routing procedures must be capable of ensuring that traffic is forwarded along routes that offer the required services without violating domain usage restrictions. IDPR meets this goal. It has been designed to accommodate an internet comprising a very large number of administrative domains with diverse service offerings and requirements. The routing process is susceptible to faults. The faults result in incorrect delivery of messages or loss of messages. The policy requirements and restrictions should also be taken into account. Self-stabilization is the most general technique to design fault tolerant systems.

4.2 Related Work

An architecture for Inter-Domain Policy Routing is specified in [34]. The specification of the Inter Domain Policy Routing Protocol is given in [36]. It presents the set of protocols and procedures that constitute Inter-Domain Policy Routing (IDPR). The key concepts and protocols developed as part of the Inter-Domain Policy Routing (IDPR) architecture are summarized in [16]. It has placed particular emphasis on the route installation and packet forwarding mechanisms.

Network topology maintenance is an important component of Internet routing. A lot of research has been done in this area [25, 37, 29]. Nodes/links failures directly cause the network topology changes, which implicitly introduce errors in routing. Since network topology maintenance protocol is the underlying protocol for most of the routing protocols, its stability is very important. The topology update problem has been discussed in [19, 22, 27, 31].

A self-stabilizing routing scheme for general networks has been given in [11]. Border
Gateway Protocol uses the path vector approach for distributing routing information and the hop by hop technique for data message forwarding. A description of self-stabilizing Border Gateway Protocol is specified in [13]. It deals with the problem of rapid fluctuation of the network reachability information. The overview of functionality and a discussion of the experiments with implementation of IDPR is given in [35]. The routing information must be distributed in a reliable manner. The strategies for fault tolerant distribution of routing information are discussed in [32]. We propose a self-stabilizing IDPR using the link state approach and source specified message forwarding.

4.3 Model and Preliminaries

4.3.1 Model

**Network Topology.** Our network is a collection of autonomous systems. Each autonomous system is a collection of networks, gateways and links managed by a single administrative authority. It is represented as $G(v,e)$, where $v$ is the set of $\{\text{host,pg, network}\}$ and $e$ is the set of links. $pg$, a subset of $v$, is the policy gateway. Virtual gateway $vg$ is a a collection of directly-connected “policy gateways” in two adjoining domains. It is represented by: $vg=\{pg_1,pg_2,...pg_m\}$, where $pg_x,pg_y$ are directly connected and belong to adjoining domains or the same domain.

**Program.** The state of a process is defined by the value of its variables. The processes represent nodes or routers. The state of a system is a vector of $n+1$ components where the first $n$ represent the state of $n$ processes, and the last component refers to the set of messages in transit in the links. In the following, we refer to the state of a process and system as a (local) state and configuration, respectively. Let a distributed protocol $P$ be a collection
of binary transition relations denoted by \(\rightarrow\), on \(\mathcal{C}\), the set of all possible configurations of the system. A *computation* of a protocol \(\mathcal{P}\) is a maximal sequence of configurations \(e = \gamma_0, \gamma_1, \ldots, \gamma_t, \gamma_{t+1}, \ldots\) such that for \(t \geq 0, \gamma_t \rightarrow \gamma_{t+1}\) (a single *computation step*), if \(\gamma_{t+1}\) exists, or \(\gamma_t\) is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no action of \(\mathcal{P}\) is enabled in the final configuration. All computations considered in this thesis are assumed to be maximal.

During a computation step, one of the following actions (*local steps*) occurs on at least one process \(p\): (1) \(p\) receives a message; (2) \(p\) executes some internal actions; (3) \(p\) sends at least one message. The set of computations of a protocol \(\mathcal{P}\) in system \(S\) starting with a particular configuration \(\alpha \in \mathcal{C}\) is denoted by \(\mathcal{E}_\alpha\). The set of all possible computations of \(\mathcal{P}\) in system \(S\) is denoted as \(\mathcal{E}\).

Each action of a process is of the form:

\[
<\text{label}> <\text{guard}> \rightarrow <\text{statement}> \\
\vdots \\
<\text{statement}>
\]

The guard of an action in the program of a process \(p\) is one of the following: a local guard of \(p\) or a receiving guard of \(p\). A local guard of \(p\) is a boolean expression involving the variables of \(p\). A receiving guard of \(p\) is of the form:

\[\text{rcv} <\text{message}_\text{type}> \text{ from } <\text{sending}_\text{channel}_\text{name}>\].

The statements of a process are of four types: assignment, sending, selection, and iteration. An assignment statement of \(p\) is of the form: \(x_p := E_p\) where \(x_p\) is a variable of \(p\) and \(E_p\) is a constant or expression of the same type as \(x_p\). A sending statement of \(p\) is of the form: send \(<\text{message}_\text{type}>\) to
A selection statement of \( p \) is of the form: \( \text{if} \ldots \text{fi} \). An iteration statement of \( p \) is of the form: \( \text{for} \ldots \text{endfor} \) or \( \text{do while} \ldots \text{od} \).

The statement of an action of \( p \) updates one or more variables of \( p \). When \( p \) executes a statement, we say that "\( p \) moves" or "\( p \) executes an action". An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning, the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

**Self-stabilizing Program.** Let \( \mathcal{L} \) be a predicate (called, *legitimacy predicate*) defined with respect to a specification (predicate) \( R \). An algorithm \( A \) is *self-stabilizing* for the specification \( R \) if (i) any computation of \( A \) starting from a configuration satisfying \( \mathcal{L} \) satisfies \( R \) (correctness) and (ii) starting from any configuration \( \in \mathcal{C} \), any computation of \( A \) reaches a configuration which satisfies \( \mathcal{L} \) (convergence) in finite steps.

### 4.3.2 Problem Specification

Our self-stabilizing IDPR algorithm ensures that every message is delivered from a source host in one autonomous system to a destination host in another autonomous system, using routes that provide user traffic with the services requested within the constraints stipulated for the domains transited.

**Specification 4.1 (Inter-Domain Policy Routing)** Given a well-constructed message \( M \) from a source node \( S \) in an autonomous system, an execution of the system satisfies the Inter-Domain Policy Routing Problem (we will call it IDPR) if the following property holds:
**Reliable Delivery:** The message $M$ from $S$ will be safely delivered at its destination in another autonomous system, using routes that provide user traffic with the services requested within the constraints stipulated for the domains transited.

This property ensures correct behavior of the algorithm. We also require our algorithm to be **self-stabilizing** as per the definition given in the previous Section.

### 4.4 Inter Domain Policy Routing Algorithm

In this section, we formally present the self-stabilizing IDPR algorithm, called $\text{IDPR}$. First, we discuss the data structures (Section 4.4.1). Next, we give an informal description of the algorithm in Section 4.4.2.

#### 4.4.1 Data Structures

**Constants:** The constants are shown in Algorithm 4.1. The constants are inputs to Algorithm $\text{IDPR}$. They are maintained by the virtual gateway protocol [36].

**Messages:** The messages are shown in Algorithm 4.2.

1. **SETUP:** Establishes a path by linking pairs of policy gateways.
2. **ACCEPT:** Signals successful path establishment.
3. **REFUSE:** Signals that the path could not be successfully established. Contains identifier of the path and the reason for refusal.
4. **ERROR:** Signals path error because of duplicate or unrecognized parameters in path setup messages. Contains identifier of the path and the reason for error.
5. **TEARDOWN:** Tears down a path.
6. **ACK:** Acknowledgment for data.
7. REPAIR: Establishes a repaired path linking pairs of policy gateways.

8. DATA: The DATA message has two formats, one for the message from host to the policy gateway and one that is sent between policy gateways. The DATA message from host has the user data, source, destination, requested services and domains to be excluded. The exclude set contains the domains to be excluded. The DATA message from policy gateway has the path id in addition to these parameters.

Variables: The routing table of each policy gateway consists of six arrays as shown in Algorithm 4.3. Array exist.paths contains the existing routes. Array exist.path.id represents the path id corresponding to the routes. The arrays exist.path.prevpg and exist.path.nextpg denote the previous and next policy gateways, respectively, on the path. Array exist.path.timestamp stores the times the paths were setup. The array exist.path.accept denotes if the path has been accepted (path setup was successful). The variable buffer is used to hold the messages.

Procedures: The procedures used in the solution are shown in Algorithm 4.3. SELECT-NEXTPG selects a reachable nextpg that satisfies domain conditions. SETUPCHECKS checks if the SETUP message received contains valid information. ROUTEGEN computes a route following the source routing scheme. This route must satisfy the requested services and respect the transit policies. SELECTREPAIRPG selects an alternate nextpg that satisfies certain reachability conditions.

4.4.2 Informal Description

Algorithm TDPR consists of two main protocols: PCP and CMTIP. As the protocol PCP is very long, to help understand the algorithm, we give an abstract and a full version.
of our solution in Sections 4.6.2 and 4.6.3, respectively. The CMTP protocol is presented in Section 4.6.4.

We will take the following approach in describing Algorithm PCP. We will try to explain all the functionalities referring to the abstract version only. However, in some cases, we refer to some variables and parameters which are missing in the abstract version of the code. In those cases, we will refer to the full version of the algorithm.

In this section, we briefly explain the behavior of our algorithm in finding a path which satisfies the policy restrictions for the data from source to destination. The main algorithm is presented as three sets of actions: receive, send, and error correction actions. In Section 4.4.2.1, we explain the process followed to route a data from a source to a destination assuming that the routing tables are correct and no faults occur during the data transmission. Then in Section 4.4.2.2, we explain how faults are detected and recovered. In Section 4.4.2.3 we will present Algorithm CMTP to handle message loss and message corruption.

4.4.2.1 Normal Behavior

First we describe the overall operation in which the data travels from source to destination. Later, we describe each of the steps in detail with reference to Algorithm PCP.

The policy gateway receives a DATA from the host and generates a policy route. The route is the set of autonomous systems to be traversed to reach the destination. This list of autonomous systems is used to choose a list of policy gateways forming a path. The policy gateway checks if a path for that route already exists in the routing table. If it exists, it sends the data in a DATA message via this path, and waits for an acknowledgment. Otherwise, the policy gateway has to set up a path. The policy gateway selects a
policy gateway in the next intermediate autonomous system and sends the SETUP message to that policy gateway. The SETUP message contains the route, requested services, constituent autonomous systems, and relevant transit policies. Using this information, an intermediate policy gateway determines whether to accept or refuse the path, and to which next policy gateway to forward the SETUP message. The next and previous hop values are set in the routing table. The SETUP message is propagated along the policy gateways until it reaches the destination. Then the policy gateway in the destination autonomous system sends an ACCEPT message, and the ACCEPT message travels towards the source. Upon receiving the ACCEPT, the source sends DATA towards the destination using the next hop information in the routing table. Upon receiving the data, the destination sends an acknowledgment message ACK towards the source.

**Route Generation:** On receiving the data from the host, Action R1 is executed. The ROUTEGEN procedure is called to generate the route. A breath first traversal of the autonomous systems is performed to reach the destination. During the traversal, the transit policies of the autonomous system are checked if they match the requested services and do not belong to the exclude set (intermediate autonomous systems to be excluded). These checks are performed in Lines 22.21 - 22.24 in Algorithm 4.23 and then added to the route array in the order of the traversal.

If there already exists a path for this route, data can be sent (Line 4.03 - 4.04 in Algorithm 4.4). We can go directly to the step Send Data, followed by the step Acknowledgment. If no path exists, a path is setup using the two steps — Path Setup and Path Accept — as described below.

**Path Setup:** A next policy gateway (nextpg) is selected by calling the SELECTNEXTPG
The SELECTNEXTPG selects the next policy gateway on a path in round-robin order from its list of policy gateways contained in the present or next virtual gateway. While selecting the next policy gateway, the policy gateway uses the information contained in the SETUP message, and information provided by the virtual gateway protocol and the intra-domain routing procedure.

The SETUP message is forwarded to the policy gateway by executing Action $S_1$. The next policy gateway receives the SETUP by executing Action $R_2$. It then performs some checks by calling the procedure SETUPCHECKS. The SETUPCHECKS procedure in Algorithm 4.22 checks for a duplicate message, checks if the current domain appears in the messages, and for the correct transit policies. It returns true if the checks fail, or false otherwise. If the checks do not fail, a next policy gateway (nextpg) is selected to forward the SETUP message (Lines 4.12 - 4.18 in Algorithm 4.4). The routing table (exist_path array) values are set accordingly. (Lines 10.13 - 10.18 in Algorithm 4.10).

**Path Accept:** By executing Action $R_2$, the SETUP message is propagated until the destination is reached, since Line 4.17 in Algorithm 4.4 enables Action $S_1$ of Algorithm 4.8. When the SETUP message reaches the destination, it responds by sending an ACCEPT message. On receiving the ACCEPT message (Action $R_3$), the policy gateway checks if it is the source. If it is a source, it sends the data. If it is not, it forwards the ACCEPT message. It uses the exist_path_prevpg values to forward the ACCEPT message following the reverse path of the SETUP message. The ACCEPT message eventually reaches the source.

**Send Data:** When the ACCEPT message reaches the source, it indicates the correct establishment of the path. Now the source starts sending the DATA message. (Line
4.24 in Algorithm 4.4). Action $R_6$ is enabled and the policy gateway receives the $DATA$. The path is checked for expiration and policy restrictions. Since the routing tables are correct, $DATA$ is sent to the next policy gateway ($exist.path.nextpg$) on its path towards the destination (Line 6.27 in Algorithm 4.6).

Acknowledgment: The $ACK$ is sent when the data reaches the destination (Line 6.13 in Algorithm 4.6), and is forwarded by the intermediate policy gateways. The $ACK$ travels in the opposite direction of the data using the $exist.path.preupg$ values (Line 7.17 in Algorithm 4.7).

4.4.2.2 Faults and Recovery

1. Errors while sending the $SETUP$ message:

On receiving the $SETUP$ message by executing Action $R_2$, the policy gateway calls the procedure $SETUPCHECKS$ (Algorithm 4.22) to perform some checks.

(a) If the $path.id$ in the $SETUP$ message matches a stored id then it is a duplicate message. The policy gateway discards the message.

(b) If the current domain does not appear in the $SETUP$ message, $ERROR$ message with reason as $no\_domain$ is sent to the previous policy gateway (Lines 21.06 - 21.11). By executing Action $R_5$, an $ERROR$ message is propagated towards the source. The source sends a $TEARDOWN$ message. By executing Action $R_7$ the resources are released (the routing table entries are removed). A new route is generated by executing Action $R_5$ and a path is setup.

(c) If the adjacent virtual gateway is not reachable, a $REFUSE$ message with reason as $no\_reach$ is sent to the previous policy gateway (Lines 21.12 - 21.21).
The previous policy gateway executes Action $R_4$ and selects an alternate next policy gateway to forward the $SETUP$ message. If there are no alternate policy gateways, the previous policy gateway sends a $REFUSE$ message with reason as $no_pg$ towards the source which releases the resources by sending a $TEARDOWN$ message (Action $R_7$). By performing Action $R_7$, a new route is generated and a path is setup.

(d) If the transit policies in the $SETUP$ message do not match that of the current autonomous system, a $REFUSE$ message with reason as $no_tp$ is sent to the previous policy gateway. By executing Action $R_4$, the previous policy gateway selects an alternate next policy gateway, to forward the $SETUP$ message. If there are no alternate policy gateways, the previous policy gateway sends a $REFUSE$ with reason as $no_pg$ towards the source which releases the resources by sending a $TEARDOWN$ message.

2. Errors while sending the $DATA$ message:

The policy gateway receives $DATA$ by executing Action $R_6$. The following checks are performed to detect the faults.

(a) If the next policy gateway is not reachable, it selects an alternate next policy gateway ($nextpg$) using the procedure $SELECTREPAIRPG$ and sends a $REPAIR$ message to the next policy gateway (Line 6.22 in Algorithm 4.6). Procedure $SELECTREPAIRPG$ is given as Algorithm 4.21. It tries to select one of the following if it exists: a peer to $nextpg$ directly connected to the current policy gateway, a peer to $nextpg$ which is connected to the peer of the current policy gateway, a peer to the current policy gateway and directly connected to $nextpg$. (If
there is no alternate policy gateway, it sends a *TEARDOWN* message towards
the originator which releases the resources). On receiving the *REPAIR* mes-
message (Action $R_0$), the policy gateway performs the setup checks using procedure
*SETUPCHECKS* and if there are no errors, selects the next policy gateway
to forward the *REPAIR* message which again forwards until the destination is
reached (Lines 7.22 - 7.26 in Algorithm 4.7). The destination sends an *ACCEPT*
towards the source (Lines 7.29 - 7.31 in Algorithm 4.7). The source can then
send the *DATA* message.

(b) If the transit policies of the autonomous system do not allow traffic to flow
through, then a *REFUSE* with *reason* as *no.pg* message is sent (Lines 6.08
- 6.10 in Algorithm 4.6). By performing Action $R_4$, the *REFUSE* message reaches
the source. The source tears down the path using a *TEARDOWN* message and
then sends a new route request and path setup (Lines 5.14 - 5.23 in Algorithm
4.5). *DATA* can then be sent as in the above case.

(c) If the current policy gateway cannot recognize the path id, then an *ERROR*
message with *reason* as *no_pathid* is sent to the previous policy gateway. The
previous policy gateway executes Action $R_5$ to receive the message. It then
selects a next policy gateway using the procedure *SELECTREPAIRPG* and sends a
*REPAIR* message to the next policy gateway. The path is repaired by executing
Action $R_9$.

(d) If the current autonomous system is a member of the *exclude* set, then an
*ERROR* with *reason* as *no_domain* is sent to the previous policy gateway (Line
6.06 in Algorithm 4.6). By performing Action $R_5$ the message is sent towards
the source which then generates a new route and sets up the path for sending

\textit{DATA}.

3. Timeout Actions

When the path life time is up, i.e, the path expires, the path is torn down (Action $E_1$). The \textit{TEARDOWN} message is sent to release the resource (Algorithm 4.8).

4.4.2.3 Control Message Transport protocol

Errors in IDPR messages can have bad effects on routing. So IDPR protocols have been designed to minimize loss and corruption of control messages. Moreover, the IDPR recipient of a control message first verifies that the message is well-formed. There are three types of \textit{CMTP} messages:

1. \textit{DATAGRAM}: Contains IDPR control messages.

2. \textit{ACKCMTP}: Positive acknowledgment in response to a \textit{DATAGRAM} message.

3. \textit{NAK}: Negative acknowledgment in response to a \textit{DATAGRAM} message.

The \textit{CMTP} operates in the following way.

\textbf{Message Transmission} The IDPR protocol, like PCP, passes a copy of the message and maximum number of transmissions allotted to \textit{CMTP}. Using the control message and parameters supplied, \textit{CMTP} constructs a \textit{DATAGRAM}. A transaction identifier is assigned to associate either an \textit{ACKCMTP} or a \textit{NAK}. The protocol also calculates the length, saves a copy, sets a retransmission timer and sends the \textit{DATAGRAM}. It expects to receive either \textit{ACKCMTP} or \textit{NAK} as a response (Lines 24.01 - 24.09 in Algorithm 4.26).
DATAGRAM Reception The policy gateway receives the DATAGRAM by executing Action $R_{11}$. It performs the CMTP validation checks on the DATAGRAM. If the DATAGRAM passes the checks, the recipient delivers the message to IDPR protocol, and sends back an ACKCMTP. If it fails to pass any of the checks, CMTP returns NAK to the sender and discards the DATAGRAM (Lines 27.10 - 27.21 in Algorithm 4.26). The corresponding send actions are in Algorithm 4.27.

ACKCMTP Reception Upon receiving ACKCMTP by performing Action $R_{12}$, the policy gateway clears the retransmission timer, discards the DATAGRAM and sends an acknowledgment to the protocol. (see Lines 24.22 - 24.29 in Algorithm 4.26).

NAK Reception NAK is received by executing Action $R_{13}$. The policy gateway first checks if the number of transmissions of the DATAGRAM has exceeded the maximum number of transmissions. If the number is exceeded, then the DATAGRAM is discarded, Otherwise it is retransmitted (Lines 24.30 - 24.44 in Algorithm 4.26).

TIMEOUT If the retransmission timer times out before receiving either ACKCMTP or NAK, then the DATAGRAM is retransmitted, provided it has not exceeded the transmission allotment. Otherwise it is discarded (Action $E_2$). Then the policy gateway informs the IDPR protocol, which may resubmit the control message back to the CMTP (Lines 25.14 - 25.16 in Algorithm 4.27).
4.5 Proof of Correctness

In this section, we will prove the correctness of the algorithm presented in Section 4.4 and show that it satisfies the specifications as defined in Section 4.3.2. We first define the legitimacy predicate. Then we prove the reliable delivery property in Section 4.5.1 and the convergence property in Section 4.5.2.

The legitimacy predicate is defined as follows:

\[ \mathcal{L}_{TPR} \equiv \text{The routing tables are correct.} \]

Reliable delivery is implemented by maintaining correct routing tables. The conditions for the correctness of the routing table are given below.

Correct Routing Tables: The routing table is correct when the following properties hold.

(P1) The entries should have valid path_ids.

(P2) The entry exist_path_nextpg should not belong to any autonomous system which is part of the set exclude.

(P3) The entries exist_path_nextpg and exist_path_preupg have intra-domain connectivity or virtual gateway reachability with the current policy gateway.

(P4) The exist_path_accept flags are true only if the ACCEPT message corresponding to the SETUP message with the same path_id has been received.

(P5) The entry exist_path_nextpg is such that its transit policies allow traffic to flow in accordance with the requested services.
4.5.1 Reliable Delivery

We will prove that if the system is in a legitimate configuration, the algorithm behaves correctly. We have assumed that no new faults occur, hence the error actions will not be invoked.

**Lemma 4.1 (Reliable Delivery)** Starting from a configuration that satisfies the legitimacy predicate, data will be safely delivered at its destination using routes that provide user traffic with the services requested within the constraints stipulated for the domains transited.

**Proof.** Upon receiving the DATA from a host, a policy gateway $i$ calls the ROUTEGEN procedure which returns the route. We need to consider two cases.

**Existing Path:** If the route is an existing one, there exists a path. The policy gateway $i$ delivers the DATA if it is the destination. If $i$ is not the destination then it has to forward the DATA by executing Action $R_3$ (Algorithm 4.6). Since the exist_path_nextpg is valid as long as the predicate $LTPR$ is true, the entries have proper connectivity and satisfy the transit policy and requested services. It checks the flag exist_path_accept, and since it is correct from the assumption, DATA is forwarded to the next policy gateway (exist_path_nextpg). Thus, DATA is forwarded correctly until it reaches the destination where it is delivered to the local host (Line 6.12 in Algorithm 4.6).

**New Path:** A new path must be setup. A SETUP message with the path id is sent to the next policy gateway exist_path_nextpg by executing Action $R_1$. The SETUP message is forwarded by using Action $R_2$ until the message reaches the destination. Then the destination sends back an ACCEPT message (Line 4.20 in Algorithm 4.4).
The policy gateway forwards ACCEPT by executing Action $R_3$ until it reaches the source. As the exist\_path\_prevpg entries are valid, the ACCEPT message is sent correctly to the source. The source, upon receiving the ACCEPT, sends DATA by performing Action $R_3$ (Line 4.24 in Algorithm 4.4) which is forwarded as described in the previous case.

\[
\]

### 4.5.2 Convergence

First we prove that starting from an arbitrary configuration, the route legitimacy predicate will be satisfied in finite time. In order to prove the convergence we consider two cases. First we assume that there is no message loss or corruption. Then we assume that there are message losses or message corruption. We then conclude the proof of correctness by showing the self-stabilizing property.

#### 4.5.2.1 No Message Loss or Corruption

We consider two cases:

(i) *No data is received for the route.* The path lives for $t_{\text{path}}$. Once this timer expires, Action $\mathcal{E}_1$ tears down the path by sending the TEARDOWN message. Thus, the routing table entries will be correct.

(ii) *There is data for the route.* We now give the proof of the properties P1 through P5 for this case.

**Property 4.1** Starting from an arbitrary configuration, property P1 eventually holds.

**Proof.** Upon receiving the DATA by Action $R_6$, the policy gateway checks if the $\text{path\_id}$ in the message belongs to exist\_path\_id array (Line 6.02 in Algorithm 4.6). If it
cannot recognize the path_id, an ERROR message with reason as no_pathid is sent to the previous policy gateway. The previous policy gateway executes Action $R_5$. It calls the procedure SELECTREPAIRPG (Algorithm 4.21) to select an alternate policy gateway. The REPAIR message is sent to the alternate policy gateway. Action $R_9$ is enabled, which repairs the path by setting the path_id variable correctly.

Property 4.2 Starting from an arbitrary configuration, property $P2$ eventually holds.

Proof. The policy gateway receives the DATA by executing Action $R_6$. If the current policy gateway belongs to an excluded autonomous system, then an ERROR message with reason as no_domain is sent to the previous policy gateway. The message is forwarded by Action $R_5$ towards the source which tears down the path. The source then generates a new route, and sends a new SETUP message to setup the correct path (see Lines 6.05 - 6.07 in Algorithm 4.6).

Property 4.3 Starting from an arbitrary configuration, property $P3$ eventually holds.

Proof. The policy gateway receives the DATA by performing Action $R_6$. If the next policy gateway (exist_path_nextpg) is not reachable, the policy gateway calls the procedure SELECTREPAIRPG to select an alternate policy gateway. The REPAIR message is sent to the alternate policy gateway. Action $R_9$ sets up the repaired path towards the destination. Thus, both the next policy gateway (exist_path_nextpg) and previous policy gateway (exist_path_prevpg) entries will be set correctly.
Property 4.4 Starting from an arbitrary configuration, property P4 eventually holds.

Proof. By executing Action $R_6$ (Algorithm 4.15), the policy gateway receives the DATA. If the flag `exist_path_accept` is not set correctly (i.e., it is false) an ERROR message with reason as `no_pathid` is sent to the previous policy gateway. The previous policy gateway executes Action $R_5$ and repairs the path as in the previous case. Thus, setting the flag `exist_path_accept` will be set correctly.

□

Property 4.5 Starting from an arbitrary configuration, property P5 eventually holds.

Proof. If the transit policies do not allow DATA to flow through, then a REFUSE message is sent to the previous policy gateway. The previous policy gateway executes Action $R_4$, and forwards the REFUSE message with `no_pg` as reason towards the source. The source tears down the path and sets up a new path.

□

Lemma 4.2 (Convergence without Message Loss) Starting from an arbitrary configuration, without considering message losses and corruption, $L_{IDPR}$ eventually holds.

Proof. The proof follows from Properties 4.1, 4.2, 4.3, 4.4, and 4.5.

□

4.5.2.2 Message Loss or Corruption

We now prove how the message loss and corruption are handled and how the routing tables will be corrected in finite time.
Lemma 4.3 (Convergence with Message Loss) Starting from an arbitrary configuration, considering message losses and corruption, $L_{IDPR}$ eventually holds.

Proof.

The IDPR protocol, like PCP, passes a copy of the message to $CMTP$. The policy gateway receives the message from local host by executing Action $R_{10}$. It constructs a $DATAGRAM$ using the message and then sends it using Action $S_1$ (Algorithm 4.27). By executing Action $R_{11}$, the next policy gateway receives the $DATAGRAM$. It performs the $CMTP$ validation checks on the $DATAGRAM$ by calling the procedure CMTPVALIDATIONCHECKS. If it fails to pass any of the checks (i.e., the message is corrupted), the policy gateway returns $NAK$ to the sender and discards the $DATAGRAM$. Upon receiving the $NAK$ by performing Action $R_{13}$, the policy gateway first checks if the number of transmissions of the $DATAGRAM$ has exceeded the maximum number of transmissions. If the number is exceeded, then the $DATAGRAM$ is discarded. Otherwise, it is retransmitted.

The message loss is dealt with in the following way. If the retransmission timer times out before receiving either $ACKCMTP$ or $NAK$, then the policy gateway executes Action $E_2$. It retransmits the $DATAGRAM$ provided it has not exceeded the transmission allotment. If it has exhausted its allotment, then it discards and informs the IDPR protocol, which may resubmit the control message back to the $CMTP$ (Lines 25.14 - 25.16 in Algorithm 4.27).

Thus, we proved that in case of message loss and corruption, all the messages will be either retransmitted or discarded. So, now the proof of the routing table correctness is the same as for Lemma 4.2.

□

Theorem 4.1 (Convergence) Starting from an arbitrary configuration, $L_{IDPR}$ eventu-
ally holds.

Proof. The proof follows from Lemmas 4.2 and 4.3.

\[ \square \]

Theorem 4.2 (Self-stabilizing) Algorithm IDPR is self-stabilizing.

Proof. Follows directly from Lemma 4.1 and Theorem 4.1.

\[ \square \]
4.6 Pseudo Code for the Algorithms

4.6.1 Data Structures (PCP)

Algorithm 4.1 Self-Stabilizing Path Control Protocol (Algorithm PCP) for Policy gateway $i$ (Constants).

1.01 Constants:

1.02 $n$: integer /* number of policy gateways */
1.03 $k$: integer /* number of autonomous systems */
1.04 $h$: integer /* number of hosts */
1.05 $bandwidth$: integer /* minimum bandwidth required */
1.06 $routedeZay$: integer /* maximum delay */
1.07 $sessioncost$: integer /* maximum cost */
1.08 $time$: integer
1.09 $e$: integer

/* The following information is obtained from the routing information database
it represents the link state */
1.10 $pgas$: array $[0..n-1]$ of $0..k$ /* AS id of policy gateways */
1.11 $as$: array $[0..h-1]$ of $0..k$ /* AS id of hosts */
1.12 $type$: array $[0..n-1]$ of $\{exit, entry\}$ /* type of policy gateway entry or exit */
1.13 $vreach$: array $[0..k-1, 0..k-1]$ of boolean /* reachability of As */
1.14 $intrareach$: array $[0..n-1, 0..n-1]$ of boolean /* reachability between policy gateways */
1.15 $vg$: array $[0..n-1]$ of $<0..k-1, 0..k-1>$ /* tp-transit policy */
1.16 $tp$.$bw$: $bandwidth$
1.17 $tp$.$rd$: $routedeZay$
1.18 $tp$.$sc$: $sessioncost$
1.19 $tp$: array $[0..k-1]$ of $<tp$.$bw$, $tp$.$rd$, $tp$.$sc$>
1.20 $path$.$lif$: $time$ /* life time of a path */
Algorithm 4.2 Algorithm PCP (Messages)

Messages:

- ACCEPT (path.id, source, destination)
  - path.id : integer
  - source, destination : 0..h - 1

- REFUSE (path.id, reason)
  - path.id : integer
  - reason = {no_py, no_reach, no_pathid, no_tp, no_domain}

- ERROR (path.id, reason)
  - path.id : integer
  - reason = {no_py, no_reach, no_pathid, no_tp, no_domain}

- TEARDOWN (path.id, direction)
  - path.id : integer
  - direction = {prev, next}

- SETUP (path.id, source, destination, rs, route, routejp, y)
  - path.id : integer
  - source, destination : 0..h - 1
  - /* rs-Requested Services */
  - rs : concatenated string of rs.bw, rs.rd, rs.sc
  - rs.bw : bandwidth
  - rs.rd : routedelay
  - rs.sc : sessioncost
  - route : array [0..k - 1] of 0..k
  - routejp : array [0..k - 1] of tpstring
  - y : integer

- ACK (path.id, source, destination)
  - path.id : integer
  - source, destination : 0..h - 1

- REPAIR (SETUP)
  - SETUP : SETUP message

- DATA (dat, path.id, source, destination, rs, exclude)
  - dat : data
  - path.id : integer
  - source, destination : 0..h - 1
  - exclude : {e | e E 0..k}
  - /* rs-Requested Services */
  - rs : concatenated string of rs.bw, rs.rd, rs.sc
  - rs.bw : bandwidth
  - rs.rd : routedelay
  - rs.sc : sessioncost

- DATA (dat, source, destination, rs, exclude)
  - dat : data
  - source, destination : 0..h - 1
  - /* exclude -Domains to be excluded */
  - exclude : {e | e E 0..k}
  - /* rs-Requested Services */
  - rs : concatenated string of rs.bw, rs.rd, rs.sc
  - rs.bw : bandwidth
  - rs.rd : routedelay
  - rs.sc : sessioncost
Algorithm 4.3 Algorithm PCP (Variables)

Variables:

- $m$: integer
- $vngsource, mydestination, g : 0..h - 1$ /* hosts*/
- $timestamp, currenttime : systemtime$
- $p, q, next, nextpg, prevpg, pg, j : 0..n - 1$ /* policy gateways */
- $path_id, mypath_id : 0..m$
- $status: array [0..n - 1] of \{good, bad\}$
- $errorflag, nextflag : boolean$
- $buffer : \{<empty>, DATA, ACCEPT, SETUP, REFUSE, ERROR, ACK, REPAIR, TEARDOWN\}$
- $exists\_path\_id : array [0..m - 1] of path\_id$
- $exists\_path\_prevpg : array [0..m - 1] of 0..n - 1$
- $exists\_path\_nextpg : array [0..m - 1] of 0..n - 1$
- $exists\_path\_timestamp : array [0..m - 1] of time$
- $exists\_path\_accept : array [0..m - 1] of boolean$

/* flags to denote start sending messages, sent messages and acknowledgments */
- $send\_setup, send\_data, send\_accept, send\_ack : boolean$
- $send\_repair, send\_refuse, send\_error, send\_tear\_prev, send\_tear\_next : boolean$

Procedures:

- $SETUPCHECKS(in prevpg, path\_id, exists\_path\_timestamp, exists\_path\_id, pgas, as, route, route\_tp, \gamma, vgreach, tp, out error\_flag, status)$
- $SELECTNEXTPG(in i, route, f, vg, pgas, status, introreach, vgreach, type, out nextpg, y)$
- $SELECTREPAIRPG(in i, route, f, vg, pgas, status, introreach, vgreach, type, out nextflag)$
- $ROUTEGEN(in i, d, vgreach, tp, rs, exclude, tp, out route, route\_tp)$
4.6.2 Abstract Version (PCP)

Algorithm 4.4 Algorithm PCP (Abstract Version Part I)

4.01 $(R_1)$ \texttt{rcv} DATA from $g$ —>  
  /* generate route */  
  ROUTEGEN;  
  \texttt{if} route \in exist_paths —>  
  \texttt{cansend, senddata} = true, true  
  \texttt{pathid} = m \mid m \notin exist_path_id;  
  /* select next pg */  
  SELECTNEXTPG;  
  \texttt{cansend, sendsetup} = true, true  
  fi

4.10 $(R_2)$ \texttt{rcv} SETUP from $j$ —>  
  /* perform setupchecks */  
  SETUPCHECKS;  
  \texttt{if} (pgas[i] \neq as[destination]) —>  
  /* select next pg */  
  SELECTNEXTPG;  
  \texttt{if} nextpg = $\emptyset$ —>  
  \texttt{cansend, sendreuse, reason} = true, true, no_pg  
  \texttt{if} nextpg = $\emptyset$ —>  
  \texttt{cansend, sendsetup} = true, true  
  fi

4.22 $(R_3)$ \texttt{rcv} ACCEPT from $j$ —>  
  \texttt{if} pgas[i] = as[source] —>  
  \texttt{cansend, senddata} = true, true  
  \texttt{if} pgas[i] \neq as[source] —>  
  \texttt{cansend, sendaccept} = true, true  
  fi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.5 Algorithm PCP (Abstract Version Part II)

5.01 (R_4) [] recv REFUSE from j ——
5.02 if ((reason = no_reach) \lor (reason = no_tp)) ——
   /* select next pg */
5.03   SELECTNEXTPG;
5.04   if (nxtpg = \emptyset) ——
5.05       if pgas[i] = as[source] ——
5.06           discard
5.07       [] pgas[i] ≠ as[source] ——
5.08           cansend, sendrefuse, reason = true, true, no.pg
5.09       fi
5.10 [] (nxtpg = \emptyset) ——
5.11       cansend, sendrepair = true, true
5.12 fi
5.13 fi
5.14 if reason = no.pg ——
5.15       if pgas[i] = as[source] ——
5.16         cansend, sendtearnew = true, true
5.17 ROUTEGEN;
5.18      SELECTNEXTPG;
5.19      cansend, sendsetup = true, true
5.20      [] pgas[i] ≠ as[source] ——
5.21         cansend, sendrefuse, reason = true, true, no.pg
5.22 fi
5.23 fi
5.24 (R_5) [] recv ERROR from j ——
5.25 if (reason = no_domain) ——
5.26       if pgas[i] ≠ as[source] ——
5.27         cansend, senderror reason = true, true, no.domain
5.28       [] pgas[i] = as[source] ——
5.29         cansend, sendtearnew = true, true
5.30 ROUTEGEN;
5.31      SELECTNEXTPG;
5.32      cansend, sendsetup = true, true
5.33 fi
5.34 [] reason = no.pathid ——
   /* unrecognized path id */
5.35       if path_id \notin exist.path_id ——
5.36         discard
5.37       [] path_id \in exist.path_id ——
   /* existing path */
5.38      SELECTREPAIRPG;
   /* cannot select nextpg teardown the path */
5.39 if ~(nextflag) ——
5.40      cansend, sendtearprev, direction=true, true, prev
5.41       [] if (nextflag) ——
5.42         cansend, sendrepair = true, true
5.43 fi
5.44 fi
5.45 fi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.6 Algorithm PCP (Abstract Version Part III)

6.01 \((R_6)\) [] rcv DATA from \(j\) —>
/* unrecognized path id */
6.02 if path.id \(\notin\) exist.path.id —>
6.03 cansend, senderror, reason = true, true, no.pathid
6.04 fi
/* check for exclude array */
6.05 if \((pgas[i] \in\) exclude) —>
6.06 cansend, senderror, reason = true, true, no.domain
6.07 fi
/* check for Transit policies */
6.08 if \(tp[pgas[i]] \neq rs\) —>
6.09 cansend, sendrefuse, reason = true, true, no.pg
6.10 fi
6.11 if \((pgas[i] = as[destination])\) —>
6.12 deliver dat
6.13 cansend, sendack = true, true
6.14 []\((pgas[i] \neq as[destination])\) —>
6.15 next = nextpg;
6.16 \(x, z = pgas[i], pgas[next]\);
6.17 if \(\neg(vreach[x, z])\) —>
6.18 SELECTREPAIRPG;
/* cannot select nextpg teardown the path */
6.19 if \(\neg(\neg(nextflag))\) —>
6.20 []\((\neg(nextflag))\) —>
6.21 cansend, sendrepair = true, true
6.22 fi
6.23 fi
6.24 []\(vreach[x, z]\) —>
6.25 if \(\neg(exist.path.accept[\neg(path.id)]\) = true
6.26 buffer = DATA;
6.27 cansend, senddata = true, true
6.28 []\(\neg(exist.path.accept[\neg(path.id)]\) = true —>
6.29 cansend, senderror, reason = true, true, no.pathid
6.30 fi
6.31 fi
Algorithm 4.7 Algorithm PCP (Abstract Version Part IV)

7.01 \((\mathcal{R}_7)\) \[ rev \text{TEARDOWN} \] \(\text{from } j \) \(\rightarrow\)

7.02 \(\forall m \mid \text{exist_path_id}[m] = \text{path_id} \) \(\rightarrow\)

7.03 \(\text{exist_path_id}[m] = 0; \)

7.04 \(\text{if } ((\text{pgas}[i] = \text{as}[\text{source}]) \lor (\text{pgas}[i] = \text{as}[\text{destination}])) \rightarrow \)

7.05 \(\text{discard} \)

7.06 \(\text{if } ((\text{pgas}[i] \neq \text{as}[\text{source}]) \land (\text{pgas}[i] \neq \text{as}[\text{destination}])) \rightarrow \)

7.07 /* propagate teardown towards originator */

7.08 \(\text{if } \text{direction} = \text{prev} \rightarrow \)

7.09 \(\text{cansend, sendteardown} = \text{true, true} \)

7.10 /* propagate teardown towards destination */

7.11 \(\text{if } \text{direction} = \text{next} \rightarrow \)

7.12 \(\text{cansend, sendteardown} = \text{true, true} \)

7.13 \(\text{fi} \)

7.14 \(\text{fi} \)

7.15 \(\text{fi} \)

7.16 \(\text{id} \)

7.17 \(\text{id} \)

7.18 \(\text{id} \)

7.19 \(\text{id} \)

7.20 /* perform setup checks

7.21 \text{SETUPCHECKS; \)

7.22 \text{id} \)

7.23 \text{id} \)

7.24 \text{id} \)

7.25 \text{id} \)

7.26 \text{id} \)

7.27 \text{id} \)

7.28 \text{id} \)

7.29 \text{id} \)

7.30 \text{id} \)

7.31 \text{id} \)

(Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.)
Algorithm 4.8 Algorithm PCP (Abstract Version Part V)

\( S_1 \)

\( (\exists \text{path}_i) (\text{exist.path.timestamp.path}_i - \text{currenttime} > \text{path}.lif) \) →

\( (\exists \text{pgas}[i] = \text{as}[source]) \) →

\( \text{buffer} = \text{TEARDOWN} \);
\( \text{cansend}, \text{sendtearnext} = \text{true, true} \)

\( (\text{pgas}[i] = \text{as}[destination]) \) →

\( \text{buffer} = \text{TEARDOWN} \);
\( \text{cansend}, \text{sendtearprev} = \text{true, true} \)

\( (\text{pgas}[i] \neq \text{as}[source] \land \text{pgas}[i] \neq \text{as}[destination]) \) →

\( \text{buffer} = \text{TEARDOWN} \);
\( \text{cansend}, \text{sendtearprev} = \text{true, true} \)

\( \text{fi} \)

\( \text{fi} \)

\( (\text{cansend}) \) →

\( \text{if} (\text{cansend}) \) →

(\( \text{senddata} \)) →

\( \text{send DATA to nextpg} \);
\( \text{senddata} = \text{false} \)

(\( \text{sendsetup} \)) →

\( \text{send SETUP to nextpg} \);
\( \text{sendsetup} = \text{false} \)

(\( \text{sendaccept} \)) →

\( \text{send ACCEPT to prevpg} \);
\( \text{sendaccept} = \text{false} \)

(\( \text{sendrefuse} \)) →

\( \text{send REFUSE to prevpg} \);
\( \text{sendrefuse} = \text{false} \)

(\( \text{senderror} \)) →

\( \text{send ERROR to prevpg} \);
\( \text{senderror} = \text{false} \)

(\( \text{sendtearnext} \)) →

\( \text{send TEARDOWN to nextpg} \);
\( \text{sendtearnext} = \text{false} \)

(\( \text{sendtearprev} \)) →

\( \text{send TEARDOWN to prevpg} \);
\( \text{sendtearprev} = \text{false} \)

(\( \text{sendack} \)) →

\( \text{send ACK to prevpg} \);
\( \text{sendack} = \text{false} \)

(\( \text{sendrepair} \)) →

\( \text{send REPAIR to nextpg} \);
\( \text{sendrepair} = \text{false} \)

\( \text{fi} \)

\( \text{cansend} = \text{false} \)
4.6.3 Full Version (PCP)

Algorithm 4.9 Algorithm PCP (Full Version Receive Data from host)

/* data message from host acts as route request message */
9.01 \( (P_1) \) rcv DATA\((\text{dat, source, destination, rs, exclude}) \) from g —> 
9.02 ROUTEGEN\((\text{in pgas[i], as[destination], vgreach, tp,}
\hspace{1em} \text{rs, exclude, tp. out route, route.tp}) \);
/* path exists already */
9.03 if route ∈ exist_paths —>
9.04 mypathid = exist_paths\[(\text{route}.\text{path.id})
9.05 nextpg = exist.path.nextpg\[(\text{mypathid})
9.06 buffer = DATA;
9.07 cansend, senddata = true, true
9.08 if (route ∈ exist_paths) —>
9.09 buffer = dat;
9.10 pathid = m | m ∈ exist.path.id;
9.11 y = 0;
9.12 f = y;
9.13 SELECTNEXTPG\((\text{in i, route, f, vg, pgas, status, intrareach, vgreach,}
\hspace{1em} \text{type, out nextpg, y})\);
9.14 exist.path.id[m + 1] = path.id;
9.15 exist.path.nextpg[path.id] = nextpg;
9.16 exist.path.preopp[path.id] = preopp;
9.17 exist.path.timestamp[path.id] = currenttime;
9.18 buffer = SETUP;
9.19 cansend, sendsetup = true, true
9.20 fi
Algorithm 4.10 Algorithm \( \text{PCP} \) (Full Version Receive Setup)

10.01 (\( R_2 \)) \( \text{rcv \ SETUP}(\text{path} \_\text{id}, \ \text{source}, \ \text{destination}, \ \text{rs}, \ \text{route}, \ \text{route}_1, \ \text{y}) \) from \( j \) \( \rightarrow \)
10.02 \( \text{errorflag} = \text{false}; \)
10.03 \( \text{prevpg} = j; \)
10.04 \( \text{SETUPCHECKS}(\text{in prevpg}, \ \text{path} \_\text{id}, \ \text{exist.path.timestamp}, \ \text{exist.path} \_\text{id}, \ \text{pgas}, \)
10.05 \( \ \text{as}, \ \text{route}, \ \text{route}_1, \ \text{y}, \ \text{vgreach}, \ \text{tp}, \ \text{out errorflag, status}); \) \( \rightarrow \)
10.06 \( \text{if} \ (\text{errorflag}) \rightarrow \)
10.07 \( \text{if} \ (\text{pgas}[i] \neq \text{as[destination]}) \rightarrow \)
10.08 \( f = y; \)
10.09 \( \text{SELECTNEXTPG}(\text{in i, route, f, pgas, status, intrareach, vgreach,} \)
10.10 \( \ \text{type, out nextpg, y}); \) \( \\
10.11 \text{if nextpg} = \emptyset \rightarrow \)
10.12 \( \text{buffer} = \text{REFUSE}; \)
10.13 \( \text{cansend, sendrefuse, reason} = \text{true, true, no.pg} \)
10.14 \( (\text{nextpg} = \emptyset) \rightarrow \)
10.15 \( \text{exist.path.id}[m + 1] = \text{path.id}; \)
10.16 \( \text{exist.path.nextpg}[\text{path.id}] = \text{nextpg}; \)
10.17 \( \text{exist.path.prevpg}[\text{path.id}] = \text{prevpg}; \)
10.18 \( \text{exist.path.timestamp}[\text{path.id}] = \text{currenttime}; \)
10.19 \( \text{buffer} = \text{SETUP}; \)
10.20 \( \text{cansend, sendsetup} = \text{true, true} \)
10.21 fi
10.22 \( (\text{pgas}[i] = \text{as[destination]}) \rightarrow \)
10.23 \( \text{nextpg} = \text{prevpg}; \)
10.24 \( \text{mydestination} = \text{source}; \)
10.25 \( \text{mysource} = \text{destination}; \)
10.26 \( \text{buffer} = \text{ACCEPT}; \)
10.27 \( \text{exist.path.accept}[\text{path.id}] = \text{true}; \)
10.28 \( \text{cansend, sendaccept} = \text{true, true} \)
10.29 fi
10.30 fi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.11 Algorithm PCP (Full Version Receive Accept)

/* mysource=destination, mydestination=originator */
11.01 (R3) [] recv ACCEPT(path_id, mysource, mydestination) from j —> 
/* propagate accept towards the originator */
11.02 if pgas[i] ≠ as[mydestination] —> 
11.03 exist_path_accept[path_id] = true;
11.04 prevpg = exist_path_preupp[path_id];
11.05 buffer = ACCEPT;
11.06 cansend, sendaccept = true, true
/* originator sends the dat */
11.07 [] pgas[i] = as[mydestination] —> 
11.08 exist_path_accept[path_id] = true;
11.09 nextpg = exist_path_nextpg[path_id];
11.10 buffer = DATA;
11.11 cansend, senddata = true, true
11.12 fi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.12 Algorithm PCP (Full Version Receive Refuse)

12.01 \( (R_4) \) [ ] rcv REFUSE(path.id, reason) from j ——
/* select alternate PG */
12.02 if ((reason = no_reach) \( \lor \) (reason = no.tp)) ——
12.03 \( f = y \mid \text{route}[y] = \text{pgas}[i] \);  
12.04 SELECTNEXTPG(in i, route, f, vg, pgas, status, intrareach, vgreach,  
type, out nextpg, y);
12.05 if nextpg = \( \emptyset \) ——
12.06 \( \text{buffer} = \text{REFUSE}; \)
12.07 cansend, sendrefuse, reason=true, true, no.pg
12.08 if nextpg = \( \emptyset \) ——
12.09 exist.path.id[m + 1] = path.id;
12.10 exist.path.nextpg[path.id] = nextpg;
12.11 exist.path.prevpg[path.id] = prevpg;
12.12 exist.path.timestamp[path.id] = currenttime;
12.13 buffer = SETUP;
12.14 cansend, sendsetup = true, true
12.15 fi
12.16 if pgas[i] \( \neq \) as[source] ——
/* No pg exists: propagate refuse to originator */
12.17 prevpg = exist.path.prevpg[path.id];  
12.18 buffer = REFUSE;
12.19 cansend, sendrefuse, reason=true, true, no.pg
12.20 if pgas[i] = as[source] ——
/* teardown the path */
12.21 buffer = TEARDOWN;
12.22 cansend, sendteardown, direction = true, true, next
12.23 ROUTEGEN(in pgas[i], as[destination], vgreach, tp,  
rs, exclude, tp, out route, route.tp);
12.24 SELECTNEXTPG(in i, route, f, vg, pgas, status, intrareach  
vgreach, type, out nextpg, y);
12.25 buffer = SETUP;
12.26 cansend, sendsetup = true, true
12.27 fi
Algorithm 4.13 Algorithm PCP (Full Version Receive Error)

13.01 (R_s) if rcv ERROR(path_id, reason) from j —–
13.02 if (reason = no domain) —–
13.03 if pgas[i] ≠ as[source] —–
13.04 prevpg = exist_path.prevpg[path_id];
13.05 buffer = ERROR;
13.06 cansend, senderror, reason=true, true, no_domain
13.07 [] pgas[i] = as[source] —–
13.08 buffer = TEARDOWN;
13.09 cansend, sendtearnext, direction=true, true, next
13.10 ROUTEGEN(in pgas[i], as[destination], vgreach, tp,
rs, exclude, tp, out route, route.tp);
13.11 SELECTNEXTPG(in i, route, j, vg, pgas, status, intrareach
vgrearch, type, out nextpg, y);
13.12 buffer = SETUP;
13.13 cansend, sendsetup = true, true
13.14 fi
13.15 [] reason = no pathid —–
/* unrecognized path id */
13.16 if path_id ∉ exist_path.id —–
13.17 discard
13.18 if path_id ∈ exist_path.id —–
/* existing path */
13.19 SELECTREPAIRPG(in i, route, f, vg, pgas, status, intrareach, vgreach,
type, out nextpg, y);
/* cannot select nextpg teardown the path */
13.20 if ¬(nextflag) —–
13.21 buffer = TEARDOWN;
13.22 cansend, sendtearp, direction=true, true, prev
13.23 if (nextflag) —–
13.24 buffer = REPAIR;
13.25 cansend, sendrepair=true, true
13.26 fi
13.27 fi
13.28 fi
Algorithm 4.14 Algorithm \( \mathcal{PCP} \) (Full Version Receive Data Part I)

\begin{verbatim}
14.01 (R_6) \[ \text{recv DATA}(dat, \text{path.id}, \text{source}, \text{destination}, rs, \text{exclude}) \text{ from } j \rightarrow \]
14.02 \( \text{error.flag} = \text{false}; \)
14.03 \( \text{nextpg} = \text{exist.path.nextpg}[\text{path.id}]; \)
14.04 \( \text{prepg} = \text{exist.path.prepg}[\text{path.id}]; \)

\(/^* \text{unrecognized path id} */\)
14.05 if \( \text{path.id} \notin \text{exist.path.id} \rightarrow \)
14.06 \( \text{error.flag} = \text{true}; \)
14.07 \( \text{buffer} = \text{ERROR}; \)
14.08 \( \text{consend}, \text{senderror}, \text{reason} = \text{true}, \text{true}, \text{no.pathid} \)
14.09 fi

\(/^* \text{check for exclude array} */\)
14.10 if \( \sim(\text{error.flag}) \rightarrow \)
14.11 if \( \text{pgas}[i] \in \text{exclude} \rightarrow \)
14.12 \( \text{error.flag} = \text{true}; \)
14.13 \( \text{status}[i] = \text{bad}; \)
14.14 \( \text{buffer} = \text{ERROR}; \)
14.15 \( \text{consend}, \text{senderror}, \text{reason} = \text{true}, \text{true}, \text{no.domain} \)
14.16 fi
14.17 fi

\(/^* \text{check for Transit policies} */\)
14.18 if \( \sim(\text{error.flag}) \rightarrow \)
14.19 if \( \text{tp}[\text{pgas}[i]] \neq rs \rightarrow \)
14.20 \( \text{error.flag} = \text{true}; \)
14.21 \( \text{status}[i] = \text{bad}; \)
14.22 \( \text{buffer} = \text{REFUSE}; \)
14.23 \( \text{consend}, \text{sendrefuse}, \text{reason} = \text{true}, \text{true}, \text{no.pg} \)
14.24 fi
14.25 fi
\end{verbatim}
Algorithm 4.15 Algorithm PCP (Full Version Receive data-part3)

14.25 if (pgas[i] ≠ as[destination]) ——
14.26 /* next pg is not reachable try to repair the path */
14.27 next = nextpg;
14.28 x, z = pgas[i], pgas[next];
14.29 if ¬(vreach[x, z]) ——
14.30 errorflag = true;
14.31 SELECTREPAIRPG(in i, route, f, vg, pgas, status, intrareach, vreach,
14.32 type, out nextpg, y);
14.33 /* cannot select nextpg teardown the path */
14.34 if ¬(nextflag) ——
14.35 buffer = TEARDOWN;
14.36 if (nextZap) ——
14.37 buffer = REPAIR;
14.38 cansend, sendrepair=true, true
14.39 fi
14.40 if ¬(errorflag) ——
14.41 /* send dat to next pg */
14.42 if exist_path_accept[Path.id] = true
14.43 buffer = DATA;
14.44 cansend, senddata = true, true
14.45 if exist_path_accept[Path.id] ≠ true ——
14.46 buffer = ERROR;
14.47 cansend, senderror, reason = true, true, no_pathid
14.48 fi
14.49 fi
14.50 fi; /* endi ≠ destination */
14.51 if (pgas[i] = as[destination]) ——
14.52 deliver dat;
14.53 buffer = ACK;
14.54 cansend, sendack=true, true
14.55 fi
Algorithm 4.16 Algorithm PCP (Full Version Receive teardown,ack)

15.01 (R₇)[j] \texttt{rcv TEARDOWN(path_id, direction) from } j \rightarrow
15.02 \forall m \mid \text{exist.path.id}[m] = \text{path.id} \rightarrow
15.03 \text{exist.path.id}[m] = 0;
15.04 \text{exist.path.prevpg[path.id]} = \emptyset;
15.05 \text{exist.path.nextpg[path.id]} = \emptyset;
15.06 \text{if } ((\text{pgas[i]} = \text{as[source]}) \lor (\text{pgas[i]} = \text{as[destination]})) \rightarrow
15.07 \text{discard}
15.08 \lbrack (\text{pgas[i]} \neq \text{as[source]}) \land (\text{pgas[i]} \neq \text{as[destination]}) \rbrack \rightarrow
15.09 \text{/* propagate teardown towards originator */}
15.10 \text{if } \text{direction} = \text{prev} \rightarrow
15.11 \text{buffer} = \text{TEARDOWN}:
15.12 \text{cansend, sendtearprev, direction=} \text{true, true, prev}
15.13 \text{/* propagate teardown towards destination */}
15.14 \lbrack \text{direction} = \text{next} \rightarrow
15.15 \text{buffer} = \text{TEARDOWN}:
15.16 \text{cansend, sendtearprev, direction=} \text{true, true, prev}
15.17 \text{fi}
15.18 \text{fi}

15.17 (R₈)[j] \texttt{rcv ACK(path_id, source, destination) from } j \rightarrow
15.18 \text{if } \text{pgas[i]} \neq \text{as[source]} \rightarrow
15.19 \text{prevpg} = \text{exist.path.prevpg[path.id]}:
15.20 \text{cansend, sendack} = \text{true, true}
15.21 \text{if } \text{pgas[i]} = \text{as[source]} \rightarrow
15.22 \text{discard}
15.23 \text{fi}
Algorithm 4.17 Algorithm \( \text{PCP} \) (Full Version Receive repair)

16.01 \((R_9)\) \[1 cv \text{REPAIR(SE\text{SET}UP)} \text{ from } f \rightarrow \]
16.02 \(\text{preuppg} = j\)
16.03 if \(\text{pgas}[i] \neq \text{as}[\text{destination}]\) ---
/* perform checks as in setup message */
16.04 \(\text{SETUPCHECKS}(\text{preuppg, path.id, exist.path.timestamp, exist.path.id, pgas,}
 as, \text{route, route.tp, } y, \text{vgreach, tp, out errorflag, status});\)
16.05 if \(\neg(\text{errorflag})\) ---
16.06 \(f = y;\)
16.07 \(\text{SELECTNEXTPG}(i, \text{route}, f, \text{vg, pgas}, \text{status}, \text{intrareach, vgreach,}
\text{type, out nextpg, } y);\)
/* path cannot be repaired */
if \(\text{nextpg} = \emptyset\) ---
16.08 \(\text{buffer} = \text{REFUSE};\)
16.09 \(\text{cansend, sendrefuse, reason=} \text{true, true, no.pg}\)
16.11 \(\neg(\text{nextpg} = \emptyset)\) ---
16.12 \(\text{exist.path.id}[m + 1] = \text{path.id};\)
16.13 \(\text{exist.path.nextpg}[path.id] = \text{nextpg};\)
16.14 \(\text{exist.path.preuppg}[path.id] = \text{preuppg};\)
16.15 \(\text{exist.path.timestamp}[\text{path.id}] = \text{currenttime};\)
16.16 \(\text{buffer} = \text{REPAIR};\)
16.17 \(\text{cansend, sendrepair=} \text{true, true}\)
16.18 fi
16.19 fi;
16.20
16.21 if \(\text{pgas}[i] = \text{as}[\text{destination}]\) ---
16.22 \(\text{nextpg} = \text{preuppg};\)
16.23 \(\text{mydestination} = \text{source};\)
16.24 \(\text{mysource} = \text{destination};\)
16.25 \(\text{buffer} = \text{ACCEPT};\)
16.26 \(\text{cansend, sendaccept=} \text{true, true}\)
16.27 fi
Algorithm 4.18 Algorithm PCP (Full Version Send Actions)

17.01 \( (S_1) \) if (cansend) \( \rightarrow \)
17.02 if (senddata)
17.03 \quad \text{send } DATA(\text{dat, path.id, source, destination, rs, exclude}) \text{ to nextpg;}
17.04 \quad \text{senddata} = \text{false;}
17.05 \text{\[\] if(sendsetup)}
17.06 \quad \text{send SETUP(\text{path.id, source, destination, rs, route, route.tp, y}) to nextpg;}
17.07 \quad \text{sendsetup} = \text{false;}
17.08 \text{\[\] if(sendaccept)}
17.09 \quad \text{send ACCEPT(\text{path.id, source, destination}) to prevpg;}
17.10 \quad \text{sendaccept} = \text{false;}
17.11 \text{\[\] if(sendrefuse)}
17.12 \quad \text{send REFUSE(\text{path.id, reason}) to prevpg;}
17.13 \quad \text{sendrefuse} = \text{false;}
17.14 \text{\[\] if(senderror)}
17.15 \quad \text{send ERROR(\text{path.id, reason}) to prevpg;}
17.16 \quad \text{senderror} = \text{false;}
17.17 \text{\[\] if(sendtearnext)}
17.18 \quad \text{send TEARDOWN(\text{path.id, direction}) to nextpg;}
17.19 \quad \text{sendtearnext} = \text{false;}
17.20 \text{\[\] if(sendtearprev)}
17.21 \quad \text{send TEARDOWN(\text{path.id, direction}) to prevpg;}
17.22 \quad \text{sendtearprev} = \text{false;}
17.23 \text{\[\] if(sendack)}
17.24 \quad \text{send ACK(\text{path.id, source, destination}) to prevpg;}
17.25 \quad \text{sendack} = \text{false;}
17.26 \text{\[\] if(sendrepair)}
17.27 \quad \text{send REPAIR(SETUP) to nextpg;}
17.28 \quad \text{sendrepair} = \text{false;}
17.29 \text{fi}
17.30 \text{cansend} = \text{false}
17.31 \text{fi}
Algorithm 4.19 Algorithm \( \mathcal{PCP} \) (Full Version TimeOut Actions)

\[
18.01 \quad (E_t) \quad \text{TIMEOUT} \left\{ \forall \text{path} \_\text{id} \mid (\exists \_\text{path} \_\text{timestamp} \_\text{path} \_\text{id} \mid \text{currenttime} > \text{path} \_\text{life}) \right\} \rightarrow
18.02 \quad \text{if} \ (pgas[i] = as[\text{source}]) \rightarrow
18.03 \quad \quad \text{buffer} = \text{TEARDOWN};
18.04 \quad \quad \text{cansend, sendtearnext} = \text{true, true}
18.05 \quad \left[ (pgas[i] = as[\text{destination}]) \rightarrow
18.06 \quad \quad \text{buffer} = \text{TEARDOWN};
18.07 \quad \quad \text{cansend, sendtearpv} = \text{true, true}
18.08 \quad \left[ (pgas[i] \neq as[\text{source}] \land pgas[i] \neq as[\text{destination}])
18.09 \quad \quad \text{buffer} = \text{TEARDOWN};
18.10 \quad \quad \text{cansend, sendtearnext} = \text{true, true}
18.11 \quad \quad \text{sendtearpv} = \text{true}
18.12 \quad \right)
\]
Algorithm 4.20 Algorithm PCP (Procedure: SELECTNEXTPG)

Procedure SELECTNEXTPG(in i, route, f, vg, pgas, status, intrareach, vgreach, type, out nextpg, y)

begin
if (type[i] = entry) →
    x, z = route[f], route[f + 1];
    p1 = ∃ pg | (vg[pg] = pgas[i])
    ∧ (status[pg] = good) ∧ (pg ≠ i) ∧
    intrareach[i, pg] ∧ up(pg));
if p1 →
    nextpg = pg | (vg[pg] = pgas[i])
    ∧ (status[pg] = good) ∧ (pg ≠ i) ∧
    intrareach[i, pg] ∧ up(pg));
y = f
fi

[] (type[i] = exit) →
    x, z = route[f], route[f + 1];
    p2 = ∃ pg | (vg[pg] = vg[i])
    ∧ (status[pg] = good) ∧ (pg ≠ i) ∧ vgreach[x, z]
    ∧ up(pg) ∧ pgas[pg] = route[f + 1]);
if p2 →
    nextpg = pg | (vg[pg] = vg[i])
    ∧ (status[pg] = good) ∧ (pg ≠ i) ∧ vgreach[x, z]
    ∧ up(pg) ∧ (pgas[pg] = route[f + 1]));
y = f + 1
fi
fi
end
Algorithm 4.21 Algorithm PCP (Procedure: SELECTIONPAIRPG)

Procedure SELECTIONPAIRPG(in i, route, f, vg, pgas, status, intrareach, vgreach, type, out nextflag)

20.01 begin
20.02 if (vg[i] ≠ vg[next]) --- /* select a peer to next */
20.03 p1 = ∃ q | ((vg[q] = vg[next]) ∧
20.04 up(q) ∧ intrareach[i, q]);
20.05 if (p1) ---
20.06 nextpg = q | ((vg[q] = vg[next]) ∧
20.07 up(q) ∧ intrareach[i, q]);
20.08 buffer = REPAIR;
20.09 cansend, sendrepair = true, true
20.10 fi
20.11 if (vg[i] = vg[next]) ---
20.12 y, z = pgas[i], pgas[next]
20.13 /* select a peer to next directly connected to i */
20.14 p1 = ∃ q | ((vg[q] = vg[next]) ∧ (pgas[q] = pgas[next]) ∧
20.15 vgreach[y, z] ∧ up(q));
20.16 if (p1) ---
20.17 nextpg = q | ((vg[q] = vg[next]) ∧ (pgas[q] = as[next]) ∧
20.18 vgreach[y, z] ∧ up(q));
20.19 buffer = REPAIR;
20.20 cansend, sendrepair = true, true
20.21 nextflag = true
20.22 fi
20.23 if ¬(nextflag) --- /* select a peer to next which is connected to peer of i*/
20.24 p2 = ∃ (p, q) | ((vg[q] = vg[next]) ∧ (pgas[q] = as[next]) ∧
20.25 (vg[i] = vg[p]) ∧ (pgas[i] = pgas[p]) ∧ intrareach[i, p] ∧
20.26 vgreach[y, z]);
20.27 if (p2) ---
20.28 nextpg = p | ((vg[q] = vg[next]) ∧ (pgas[q] = as[next]) ∧
20.29 (vg[i] = vg[p]) ∧ (pgas[i] = pgas[p]) ∧ intrareach[i, p] ∧
20.30 vgreach[y, z]);
20.31 buffer = REPAIR;
20.32 cansend, sendrepair = true, true
20.33 nextflag = true
20.34 fi
20.35 if ¬(nextflag) --- /* select a peer to i and directly connected to next */
20.36 p3 = ∃ p | ((vg[p] = vg[next]) ∧ (pgas[p] = pgas[next]) ∧
20.37 intrareach[i, p] ∧ vgreach[y, z]);
20.38 if (p3) ---
20.39 nextpg = p | ((vg[i] = vg[p]) ∧ (pgas[i] = pgas[p]) ∧
20.40 intrareach[i, p] ∧ vgreach[y, z]);
20.41 buffer = REPAIR;
20.42 cansend, sendrepair = true, true;
20.43 nextflag = true
20.44 fi
20.45 fi
20.46 end
Algorithm 4.22 Algorithm PCT (Procedure: SETUPCHECKS)

Procedure SETUPCHECKS(in prevpg, path.id, exist.path.timestamp, exist.path.id, pgas, as, route, route.tp, y, voreach, tp, out error.flag, status)

21.01 begin
   /* check for duplicates */
   21.02 if path.id \in exis.path.id 
      21.03 error.flag = true;
   21.04 discard
   21.05 fi;
   /* check if current domain */
   21.06 if !error.flag 
      21.07 if pgas[i] \neq route[y] 
        21.08 error.flag = true;
        21.09 send ERROR(path.id, no.domain) to prevpg
      21.10 fi
   21.11 fi;
   /* check for VG reachability */
   21.12 if !error.flag 
      21.13 if pgas[i] \neq as[destination] 
        21.14 x, z = route[y], route[y + 1];
        21.15 if !up(voreach[x, z]) 
          21.16 error.flag = true;
          21.17 status[i] = bad;
          21.18 send REUSE(path.id, no.reach) to prevpg
        21.19 fi
      21.20 fi
   21.21 fi;
   /* check for Transit policies */
   21.22 if !error.flag 
      21.23 if tp[pgas[i]] \neq route.tp[y] 
        21.24 error.flag = true;
        21.25 status[i] = bad;
        21.26 send REUSE(path.id, no.tp) to prevpg
      21.27 fi
   21.28 fi
   /* check for exclude array */
   21.29 if !error.flag 
      21.30 if (pgas[i] \in exclude) 
        21.31 error.flag = true;
        21.32 status[i] = bad;
        21.33 send ERROR(path.id, no.domain) to prevpg
      21.34 fi
   21.35 fi
   21.36 end

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.23 Algorithm FCP (Procedure ROUTEGEN Part I)

Procedure ROUTEGEN(in i, d, vgreach, tp, rs, exclude, tp, out route, route_fp)

22.01 local
22.02 level : 0..k
22.03 parent : array [0..k - 1] of 0..k - 1
22.04 dist : array [0..k - 1] of 0..k - 1
22.05 done : array [0..k - 1] of boolean
22.06 p, q : 0..k
22.07 r, temp : 0..k - 1

22.08 begin
22.09 /*Initialize array done to false*/
22.10 p = 0;
22.11 do p < k —->
22.12 done[p], p = false, p + 1
22.13 od

22.14 /* Define parent and dist of node i */
22.15 level, parent[i], dist[i], done[i] = 0, i, 0, true;

22.16 /* Define parent and dist of every other node p */
22.17 level = 1;
22.18 do level < k —->
22.19 p = 0;
22.20 do p < k —->
22.21 if done[p] —->
22.22 /* Node p belongs to previous level */
22.23 skip
22.24 [—]—done[p] ——>
22.25 /* Node p belongs to current level */
22.26 q = 0;
22.27 do q < k —->
22.28 if ((done[q] ∧ dist[q] = level - 1 ∧
22.29 vgreach[p, q] ∧ tp[p] = rs ∧
22.30 p ≠ exclude) ——>
22.31 parent[p], dist[p] = q, level;
22.32 done[p], q = true, k
22.33 [—]—(done[q] ∧ dist[q] = level - 1 ∧
22.34 vgreach[p, q] ∧ tp[p] = rs ∧
22.35 p ≠ exclude) ——> q = q + 1
22.36 fi
22.37 od
22.38 fi; p = p + 1
22.39 od; level = level + 1
22.40 od;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.24 Algorithm PCR (Procedure: ROUTEGEN Part II)

/* compute route */
22.29 if ¬done[d] ——
22.30 /* there is no policy route */
22.31 route[d] = pgas[i] + k 1
22.32 end
22.33 if done[d] ——
22.34 /* there is a route */
22.35 r = dist[d];
22.36 route[r], route[r + 1] = d, d;
22.37 do route[r] ≠ i ——
22.38 temp = parent [route[r]];
22.39 route[r - 1], r = temp, r - 1;
22.40 end
22.41 end
Algorithm 4.25 Control Message Transport Protocol (Algorithm CMTP) for Policy gateway i (Part I)

23.01 Messages:
23.02 IDPRMESSAGE(PCPMESSAGE, numofretransmissions)
23.03 PCPMESSAGE = \{DATA, ACCEPT, SETUP, REFUSE, ERROR, ACK, REPAIR, TEARDOWN\}
23.04 numofretransmissions : integer
23.05 DATAGRAM(IDPRMESSAGE, trans.id, length, auth.value)
23.06 length : integer
23.07 auth.value : integer
23.08 trans.id : 0..t − 1 /* transaction id */
23.09 NAK('nak', trans.id, length, auth.value)
23.10 length : integer
23.11 auth.value : integer
23.12 trans.id : 0..t − 1 /* transaction id */
23.13 ACKCMTP = ('ack', trans.id, length, auth.value)
23.14 length : integer
23.15 auth.value : integer
23.16 trans.id : 0..t − 1 /* transaction id */

23.17 Variables:
23.18 t, m : integer /* transaction id */
23.19 length : integer
23.20 auth.value : integer
23.21 numofretransmissions, maxtransmit : integer
23.22 errorflag : boolean
23.23 pg : 0..n-1 /* policy gateway */
23.24 exist.trans.id : array[0..m − 1] of 0..t − 1
23.25 exist.datagram : array[0..t − 1] of DATAGRAM
23.26 exist.datagram.numofretransmit : array[0..t − 1] of integer
23.27 dg : DATAGRAM
23.28 sendnow, send dg, cansend.ack, cansend.nak, errorflag, dg.sent : boolean

23.29 Procedures:
23.30 CMTPVALIDATIONCHECKS(in DATAGRAM, out errorflag)
Algorithm 4.26 Algorithm CMTP (Part II)

24.01 (R_{10}) rcv IDPRMESSAGE from localhost →
24.02 trans_id = t | t \notin exist_trans_id;
24.03 length = LENGTH(Message);
24.04 auth.val = AUTHy_VALUE(Message);
24.05 DATAGRAM = (Message, trans_id, length, auth.value);
24.06 exist_datagram[trans_id] = DATAGRAM
24.07 exist_datagram_numofretransmit[trans_id] = numofretransmissions;
24.08 send dg = true;
24.09 sendnow = true

24.10 (R_{11}) [] rcv DATAGRAM from pg →
24.11 CMTPVALIDATIONCHECKS();
24.12 if (error flag) →
24.13 NAK = ('nak', trans_id, length, auth.value)
24.14 cansend_nak = true;
24.15 sendnow = true;
24.16 []-(error flag) →
24.17 ACKCMTP = ('ack', trans_id, length, auth.value)
24.18 deliver IDPRMESSAGE to localhost
24.19 cansend_ack = true;
24.20 sendnow = true;
24.21 fi

24.22 (R_{12}) [] rcv ACKCMTP from pg →
24.23 CMTPVALIDATIONCHECKS();
24.24 if (error flag) →
24.25 discard
24.26 []-(error flag) →
24.27 ack_rcvd = true;
24.28 discard dg | dg = exist_datagram[trans_id]
24.29 fi

24.30 (R_{13}) [] rcv NAK from pg →
24.31 CMTPVALIDATIONCHECKS();
24.32 if (error flag) →
24.33 discard
24.34 []-error flag) →
24.35 NAK_rcvd = true;
24.36 dg = exist_datagram[trans_id]
24.37 maxtransmit = exist_datagram_numofretransmit[trans_id]
24.38 if current_transmission > maxtransmit →
24.39 discard dg
24.40 []current_transmission < maxtransmit →
24.41 send dg = true;
24.42 sendnow = true;
24.43 fi
24.44 fi
Algorithm 4.27 (Algorithm CMTP) (Part III)

25.01 \((S_2)\)  if (sendnow) 
25.02 \(\rightarrow\)  if (send\_dg) 
25.03 \(\rightarrow\)  send DATAGRAM
25.04 \(\rightarrow\)  \(dg\_sent=true;\)
25.05 \(\rightarrow\)  \(send\_dg=false;\)
25.06 \(\rightarrow\)  \(current\_transmission = current\_transmission + 1 ;\)
25.07 \(\rightarrow\)  if (send\_ack) 
25.08 \(\rightarrow\)  send ACK CMTP to pg
25.09 \(\rightarrow\)  \(send\_ack=false;\)
25.10 \(\rightarrow\)  if (send\_nak) 
25.11 \(\rightarrow\)  send NAK to pg
25.12 \(\rightarrow\)  \(send\_nak=false;\)
25.13 \(\rightarrow\)  fi
25.14 \((S_3)\)  \(\rightarrow\)  \(TIMEOUT(dg\_sent \land \neg (ack\_rcvd \lor nak\_rcvd))\)
25.15 \(\rightarrow\)  \(current\_transmission = current\_transmission + 1 ;\)
25.16 \(\rightarrow\)  send DATAGRAM

Algorithm 4.28 (Algorithm CMTP) (Part IV)

26.01 Procedure CMTPVALIDATIONCHECKS(in DATAGRAM, out error\_flag)
26.02 Check version type, message type
26.03 Check authentication value
26.04 Check message length
26.05 Check to recognize IDPR protocol type in header
26.06 If any of the checks fail set error\_flag=false
CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis research, we have studied a policy routing protocol that can be used for communication between different autonomous systems. We proposed a self-stabilizing Inter Domain Policy Routing algorithm. Our solution deals with faults such as message loss, message corruption, and memory corruption. Detection and recovery from faults were implemented using the paradigm of self-stabilization.

The path setup procedure was implemented to set up a successful path, respecting the transit policies. The requested services were also provided at the same time. Path failure and recovery were handled.

The research initiated in this thesis can be explored further. The route generation procedure can be improved by assigning different weights to the different metrics (such as delay, bandwidth) and by assigning weights to the routes and selecting the best route.

The domain partitions can be taken into account. The algorithm can be made to continue to operate properly in the presence of partitioned autonomous systems. Changes in transit policies configured for an autonomous system can be accommodated. Policy gateways can be used to maintain standby alternate paths that can become the primary path if necessary. Policy gateways can be made to forward messages along a path prior to confirmation of successful path establishment.
BIBLIOGRAPHY


VITA
Graduate College
University of Nevada, Las Vegas

Venkatakrishnan Tricky Ramasubramanian

Home Address:
4224 Chatham Circle, Apt#1
Las Vegas, NV 89119

Degrees:
Bachelor of Science, Computer Science, 2001
Bharathiar University, India

Thesis Title: Self-stabilizing Inter-Domain Policy Routing

Thesis Examination Committee:
Chairperson, Dr. Ajoy K. Datta, Ph.D.
Co-Chairperson, Dr. Sébastien Tixeuil, Ph.D.
Committee Member, Dr. Wolfgang Bein, Ph.D.
Committee Member, Dr. John T. Minor, Ph.D.
Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.