INTER-DOMAIN QOS ROUTING FOR GRIDSTAT

By

QING CHANG

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2013

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of QING CHANG find it satisfactory and recommend that it be accepted.

Carl H. Hauser, Ph.D., Chair

David E. Bakken, Ph.D.

Anantharaman Kalyanaraman, Ph.D.

ACKNOWLEDGMENT

First of all I would like to take this opportunity to express my sincere gratitude and thanks to Dr. Carl Hauser for his guidance and support for the past two years. He always give me professional knowledge and motivating throughout the thesis.

I would also like to thank my committee members Dr. David Bakken and Dr. Ananth Kalyanaraman for their insightful and valuable advice and comments on the thesis.

I feel grateful to Dave Anderson, Thoshitha Gamage and other members of GridStat and GridCloud group for their consistent suggestions. I also appreciate all the help and support received from the School of Electrical Engineering and Computer Science in WSU. Moreover, I am grateful to Dr. K.C Wang for his guidance in the first two years of my professional study.

Finally I would like to thank my wife Chen, my parents Shenyi, Junyan and all of my friends for their support and encouragement.

Inter-Domain QoS Routing for GridStat

Abstract

by Qing Chang, M.S.
Washington State University
August 2013

Chair: Carl Hauser

GridStat, which provides data delivery service with end-to-end Quality of Service (QoS) guarantee, is a status dissemination middle-ware framework for power grid infrastructure. It consists of a *management plane* and a *data plane*. The management plane maintains and controls the network state while the data plane delivers data.

The current routing framework forms a hierarchical tree structure in the management plane. The centralized architecture performs well in the case where subscriptions are mainly confined to lower levels of the management hierarchy. However, as the inter-domain paths computations must rely on the full network topology, the architecture obviously does not scale well. It also suffers from single-node failure. If a broker crashes, all its children brokers fail to establish the inter-domain paths.

Based on these observations, a peer-to-peer (P2P) management plane inter-domain routing framework is presented. The framework is composed of two protocols: *path vector protocol* and *path establishment protocol*. Path vector protocol is used for routing information exchange in the management plane, while path establishment protocol facilitates inter-domain disjoint paths computation. With the P2P framework, it allows both flexible topological design and new capability of fault tolerance in the management plane, as well as enabling the computation of the inter-domain disjoint paths in a distributed fashion.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

1.1 GridStat

GridStat [1] is a status dissemination middle-ware framework for power grid infrastructure. It provides data delivery service with end-to-end *quality of service* (QoS) guarantee. GridStat consists of two related networks: the *management plane* and the *data plane*.

The management plane has several functionalities: access control, security, and most importantly, comprehensive control over network state of the data plane for QoS purpose. The data plane, as its name implies, carries overall data traffic for GridStat. The management plane consists of *QoS brokers* responsible for controlling and managing the network state while the fundamental entities in data plane are *forwarding engines* responsible for data delivery service.

Routing in GridStat is the central topic in this thesis. The routing information exchange routine, path computation and allocation are manipulated by QoS brokers. Once a routing path is calculated and a path connection is established, the data stream passes through forwarding engines in the data plane. The routing architecture in current management plane is in the form of centralized tree structure. Although the centralized architecture does work well under some circumstances, it unavoidably suffers from lack of scalability and fault tolerance.

As a result, an alternative distributed peer-to-peer architecture for the management plane is proposed, as well as the corresponding protocols and mechanisms which collectively guarantee end-to-end QoS.

1.2 Routing in GridStat

From discussion above we learn that GridStat uses out-of-band routing information exchange. Out-of-band implies that a completely separate network or mechanism is used to exchange routing information from the communication network where the user traffic is carried [2]. This is the first and fundamental distinction from usual IP network where routing information exchange is performed in-band. In-band implies that control messages and data packets all share the same underlying network.

The second distinction from usual IP network lies in the fact that routing in GridStat takes place at application level since GridStat is an overlay network on top of network layer while routing in usual IP network occurs in network layer.

The third distinction is that GridStat provides QoS guarantee rather than best-effort service in usual IP network. Simple and fast, the best-effort service is widely deployed in Internet. Nevertheless it lacks precise control for necessary QoS properties such as bandwidth and delay. Therefore integrated services architecture (int-serv) is recommended for GridStat, which permits functionalities for real-time services with bounded delay and other QoS constraints.

1.3 Problem Scope

The goal of this thesis is to develop a *Management Plane P2P Routing Architecture* to address the issues of domain confidentiality, single node failure and load balancing in computing the inter-domain subscription paths. The P2P routing architecture focuses on the mechanism to exchange network information and computation of inter-domain subscription paths in a cooperative fashion.

The scope is limited in the routing functionality in the management plane. The main operational entities are management plane QoS brokers. The data plane entities remain untouched such

as publishers, subscribers, and forwarding engines.

1.4 Contribution

The contributions of this thesis are the design and development of:

- A peer-to-peer routing framework for GridStat with end-to-end QoS guarantee.
- A multi-domain multicast QoS routing protocol which involves the following two sub-protocols.
    1. Enhanced path vector protocol. The protocol not only defines the message format and broker operations for locating each other in management plane, it also incorporates publication information and data plane *aggregation representation* in its messages.
    2. Path establishment protocol. This protocol defines the message format and broker operations for inter-domain subscription path establishment.

1.5 Thesis Outline

The remainder of thesis is organized as follows. Chapter 2 introduces background knowledge and related work of QoS and GridStat routing; Chapter 3 discusses the inter-domain management framework of GridStat. Chapter 4 proposes the signaling protocol for delay-bounded disjoint path establishment; Chapter 5 discusses the design and implementation of simulation and corresponding performance analysis; Chapter 6 concludes the thesis and the future work is presented.

CHAPTER TWO

BACKGROUND AND RELATED WORK

2.1 QoS routing in Internet

Establishing inter-domain paths with QoS constraints has incurred considerable research interest in recent years. The Internet community has proposed Multiprotocol Label Switching (MPLS) [3–5]/General Multiprotocol Label Switching (GMPLS) [6, 7] mechanisms to address issues with respect to QoS routing. The routing decisions are based on labels rather than destination IP network addresses. The aim of MPLS/GMPLS is to establish label-switched paths (LSPs) for which a signaling protocol such as RSVP-TE [5, 8–15] is used.

Considering that GridStat is based on label-based routing, the mechanism presented in this thesis is inspired by the Path Computation Element (PCE) [16, 17] architecture. The mechanism and technique regarding label-switched paths finding mechanism is applied to GridStat routing with appropriate modification.

2.2 Centralized GridStat

2.2.1 Architecture

The current management plane is in the form of hierarchical tree structure. The main active entities are publishers, subscribers, status routers (forwarding engines), and QoS brokers. The main passive entities are status variables and QoS Policies [2]. In the next two sections, we will examine each of these active and passive entities.

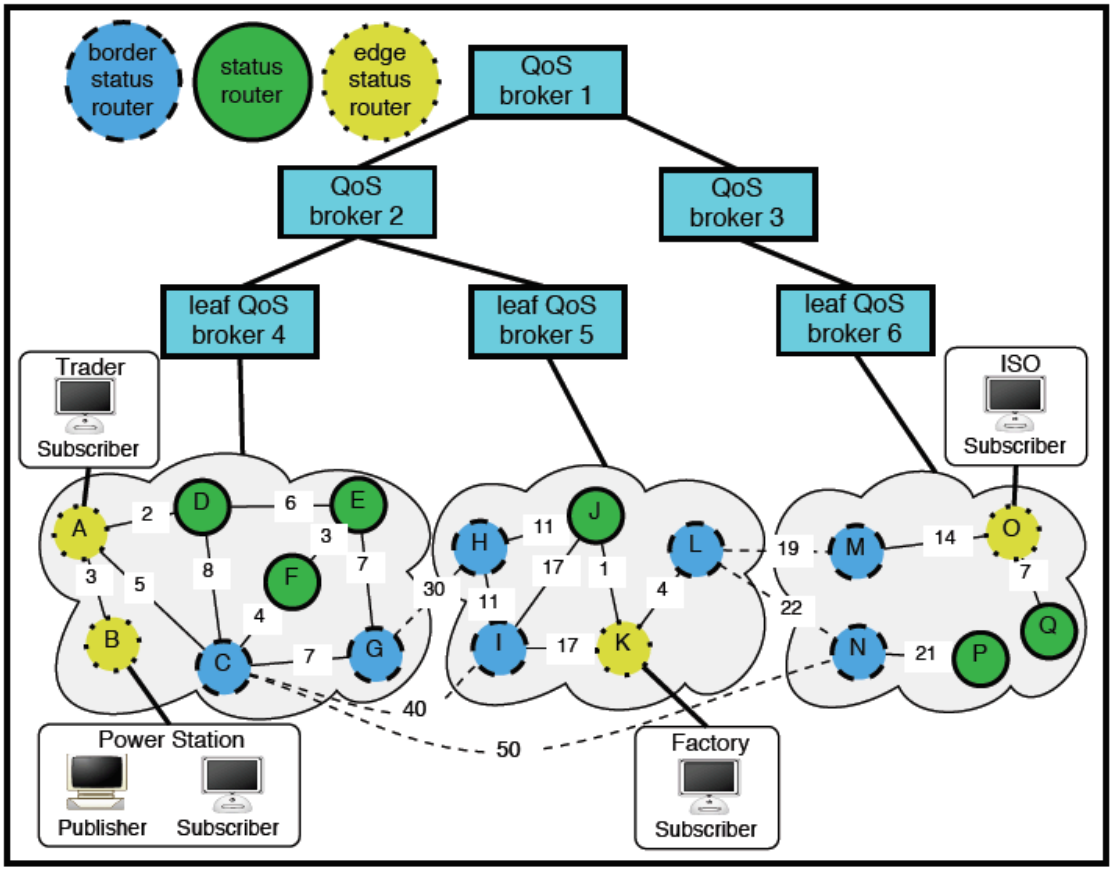

Figure 2.1: GridStat Centralized Architecture

2.2.1.1 Management Plane

A **leaf QoS broker** administers and maintains the network state of a predefined domain which is composed of a set of forwarding engines, publishers, and subscribers. A domain is also termed as a *Cloud*. Aware of the topology of its host domain, a leaf QoS broker is responsible for network resource management, in particular, monitoring, allocating and reserving network resources through issuing commands to forwarding engines. In addition, it plays pivotal role in some other aspects such as intra-domain routing path calculation and establishment and registration (un-registration) of status variable publications and subscriptions.

Non-leaf QoS brokers including the root QoS broker constitute the hierarchical structure for QoS management [18]. All the leaf QoS brokers are at the bottom level. The non-leaf QoS brokers are capable of supervising multiple domains and computing inter-domain paths according to connection requests as well as allocating resources once connections are built among domains.

A **status variable** is the data carried and delivered in GridStat. It represents state change over time on some device. It has both static and dynamic attributes. By *static* we mean that the attributes are associated with publisher name, status name and rate while we often relate *dynamic* with derived attributes, such as moving average, rate of change, etc.

2.2.1.2 Data Plane

Publishers are the source of events in the pub/sub model. They are connected to edge forwarding engines. A publisher declares what status variables it publishes along with their QoS properties.

Subscribers are the destinations of events in the pub/sub model. They are connected to edge forwarding engines. A subscriber declares what status variable it subscribes to along with desired QoS properties.

The data plane is composed of a set of **forwarding engines** which transfer data across the network much as IP routers forward packets in usual TCP/IP network. The primary difference is that forwarding engines do not make routing decisions. In other words they only forward data. The routing path and the connection establishment are both managed by QoS brokers according to specific routing protocols and algorithms. The overall structural GridStat is illustrated in Figure 2.1.

2.2.2 Routing in centralized GridStat architecture

2.2.2.1 GridStat Routing Characteristics and QoS Requirements

In general, routing in GridStat falls into the category of QoS Routing. To guarantee service qualities, certain QoS properties must be fulfilled, such as bandwidth, delay, jitter and packet loss rate, etc. The multi-constraint path-finding problem is NP-complete even in a single domain [19–21]. In addition, reliable routing is another necessary requirement in GridStat, specifically reliable routing is referred to as finding end-to-end disjoint paths with QoS properties. The problem of multi-constraint QoS disjoint path routing is also NP-hard. Irava [22] proposed multicast routing heuristics in a single domain in which a computation entity is aware of the full network topology. The proposed solution in this thesis is inspired by Irava's work with necessary modification to handle the inter-domain case. We focus on single additive property, delay. Thus multi-constraint, inter-domain QoS routing algorithm is out of scope in this thesis.

2.2.2.2 Centralized Routing Illustration

In GridStat hierarchy architecture explained in a previous section, we now consider two path finding procedures, intra-domain and inter-domain respectively, as illustrated in the Figures 2.2, 2.3. (taken from [2])

If a responsible publisher and a requesting subscriber belong to the same domain, a feasible intra-domain path is configured by the leaf QoS broker. Considering another more complicated case, if a subscriber request crosses two or more domains, the leaf QoS broker in the subscriber's domain has to forward the request upwards in the hierarchical broker tree to a non-leaf QoS broker overseeing domains involving both ends (the requesting subscriber and the destination publisher). From then on the routing algorithm can be carried out. (Figure 2.3)

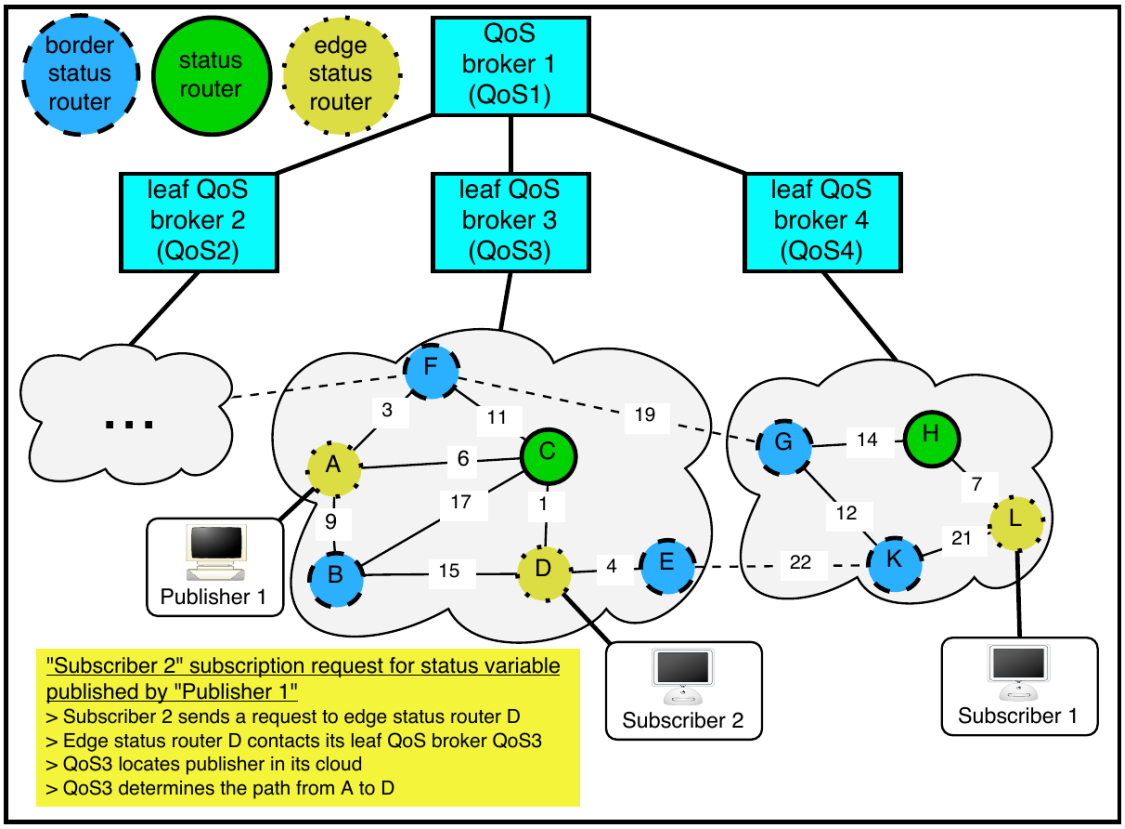

Figure 2.2: Centralized Intra-domain routing

The advantage of the current routing decision procedure in such a centralized QoS routing architecture is that path finding and connection establishment are efficient, and message overhead is fairly low. This routing architecture performs well for a small scale network. Meanwhile its drawback is equivalently apparent. As the inter-domain path computations must be made at the level where the non-leaf QoS broker has the full end-to-end visibility, the architecture obviously does not scale well in a large network. It also suffers from single node failure. Since there is no node backup mechanism in the current GridStat management plane, if any non-leaf broker crashes, it will affects the inter-domain path finding of all its children domains. In the worse case, if the root broker fails, all the inter-domain path request will be failed.

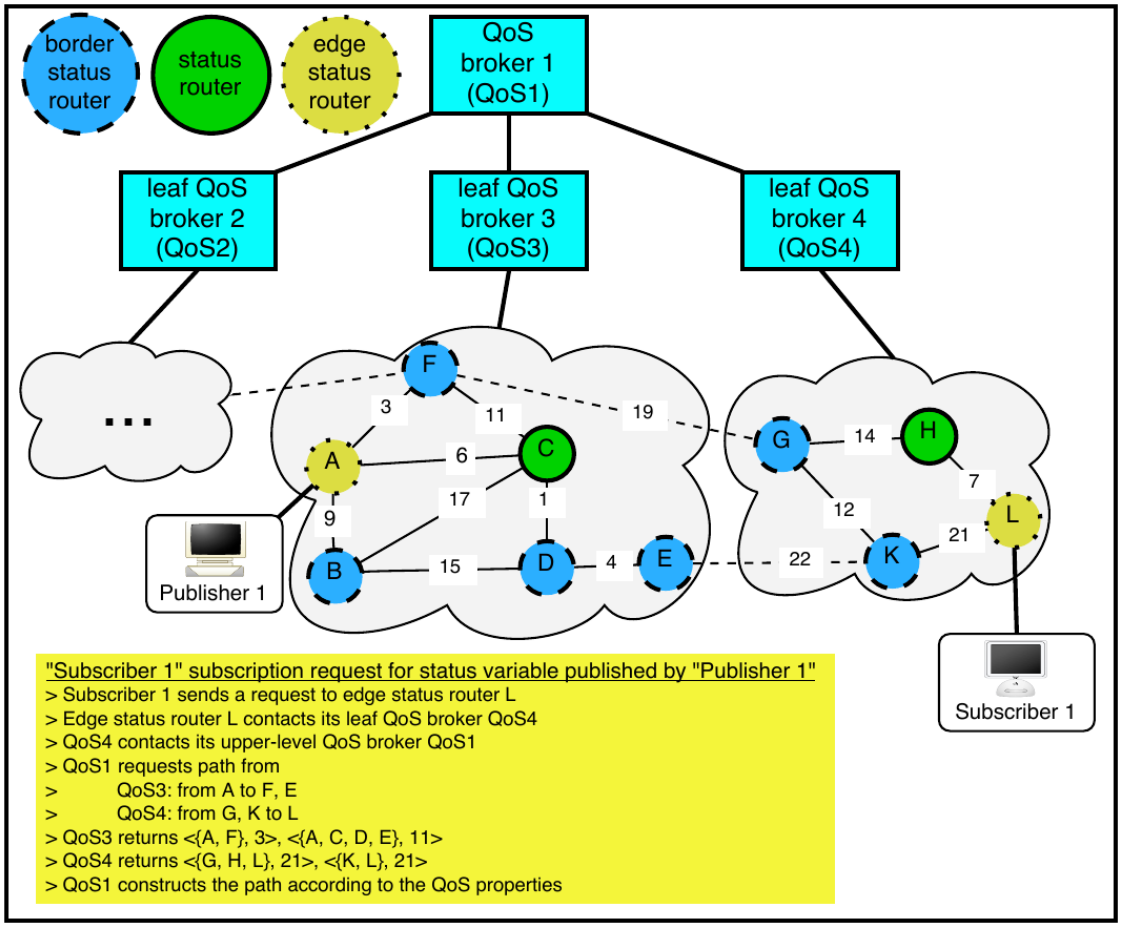

Figure 2.3: Centralized Inter-domain routing

A peer-to-peer architecture for the management plane could improve the system scalability. However this distributed approach may worsen the situation in some other aspects. For instance it is impossible for every domain to maintain the network state at every instant due to scalability, administrative, and confidentiality considerations. The requirement to compute disjoint paths makes it even harder. The good news is that a significant amount of effort has been made in recent years to resolve these issues. We will examine some of them in the following section.

2.3 End-to-End Inter-Domain Disjoint Path Approach

Here we present two proposed schemes for inter-domain disjoint path computation.

2.3.1 RSVP-TE Based

Resource Reservation Protocol (RSVP) is a connection setup protocol for a packet network [2]. Resource Reservation Protocol - Traffic Engineering (RSVP-TE) [3] is the extension of RSVP for traffic engineering. The role of traffic engineering is to optimize an operational network so that performance requirements are met, yet network resources are well utilized [2]. It generally involves establishment of MPLS label switched paths (LSPs) to achieve attributes such as bandwidth, number of hops, etc. A typical RSVP signaling procedure starts with a PATH message which flows from source to destination. It then follows with a RESV message from destination to source. The head-end (source) node can designate an explicit route by encoding the network elements to be traversed into a special object *explicit route object* (ERO). The ERO can be strict and loose depending on whether or not the list is complete. [23]

There are two RSVP-TE based mechanisms to find inter-domain disjoint paths. The first scheme is based on *primary path route object* (PPRO) [23,24]. For each domain, the ingress *border node* (BN) which receives a PATH message computes the shortest path segment to the destination on the ingress BN of the next domain is reached. The ERO in the PATH message is used for allocating downstream labels along the computed path segment. (shown in Figure 2.4 arrow 1). Next the *record route object* (RRO) in a RESV message which collects the node-level path flows upstream to the head-end node (shown in Figure 2.4 arrow 2). When RESV message arrives at the head-end node, the secondary LSP starts to compute and followed by the installation process in which the PPRO is included in the PATH message. With the awareness of the PPRO, a ingress BN which receives the PATH message computes the secondary path segment over a reduced graph

where the elements listed in the PPRO are removed together with other shared elements. Therefore the primary LSP is explicitly excluded during the secondary path installation. (arrow 3 in Figure 2.4). The drawback is that the secondary path is established subsequent to the primary path so in some cases a pair of diverse paths exist in network graph, but the subsequent computation fails to find them [23].

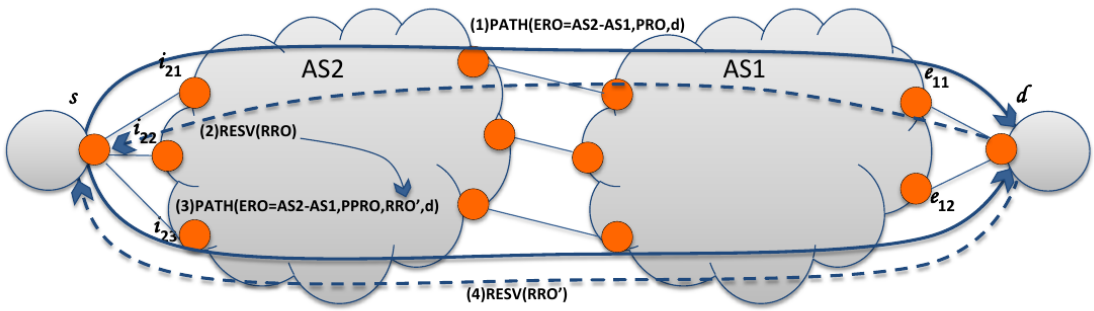

Figure 2.4: PPRO-based

The second scheme uses an *associated route object* (ARO) in PATH message to record disjoint path information. The significant improvement to the PPRO approach is that compared to the sequential computation of primary and secondary paths in PPRO, this approach computes both paths simultaneously. Let us take a look at the example illustrated in Figure 2.5. In domain AS2, when ingress BN i_1^2 receives PATH message, it computes path $p_1^2 = i_1^2, ..., e_1^2, i_1^1$ and path $p_2^2 = i_3^2, ..., e_3^2, i_2^1$ at the same time by the well-known disjoint paths algorithm [25]. p_1^2 is then encoded into ERO and p_2^2 is encoded into ARO. BN i_1^2 then initializes the primary LSP installation according to the ERO while the ARO remains unchanged. The PATH message then arrives AS1 ingress BN i_1^1, it repeats the previous process, namely, installs LSP p_1^1 and store p_2^1 to the ARO. Therefore the ARO has been extended in the progress of the primary path installation. On receiving the PATH message in destination *d*, the secondary path has been completely computed and collected

in the ARO. It is then integrated into RESV message and send back to the head-end node. At this point, source node has the complete disjoint path information and initializes the secondary path installation procedure.

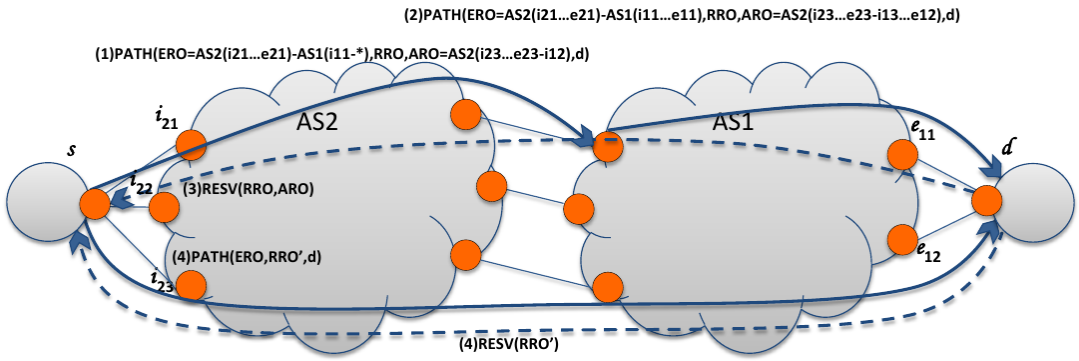

Figure 2.5: ARO-based

The drawback of RSVP-TP based approach is that it does not guarantee to find globally optimal pair of disjoint paths due to the insufficiency of inter-domain level information, specifically, the lack of coordination of egress border nodes between neighboring domains.

2.3.2 Backward-Recursive PCE-Based Computation Procedure

Backward-Recursive PCE-Based Computation Procedure (BRCP), proposed in RFC5441 [26], guarantees to find a multi-domain shortest (single) path in polynomial time. The procedure performs a exclusively search with predetermined sequence of domains. In other words, in order to obtain the optimal disjoint paths pair, we must specify a chain of domains in the first place, $domain^s, ..., domain^i, ..., domain^t$. The collaborative communication between PCEs plays an important role in enabling disjoint paths computation. The procedure starts from destination domain $domain^t$ towards source domain $domain^s$. For each domain along the predefined sequence, a Vir-

tual Shortest Path Tree (VSPT), which involves shortest paths from ingress nodes to the destination, is computed and sent to the upstream domain, as shown in Figure 2.6.

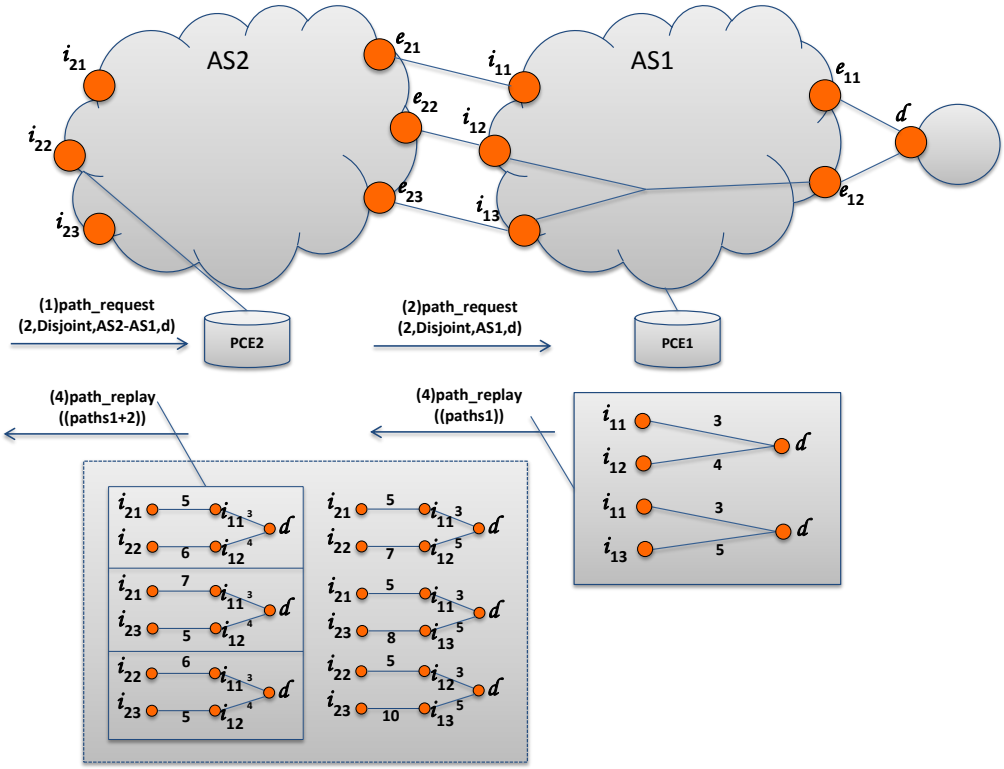

Figure 2.6: Backward-Recursive PCE-Based Computation

An extension of BRCP is proposed in RFC6007 [27] which allows finding multi-domain optimal disjoint paths. VSPT then is required to be extended to contain potential path information. More precisely, VSPT contains virtual links between each possible pair of ingress border nodes and destination for a total of B(B-1)/2 paths where B is number of border nodes [23].

The limitation of BRPC is that a predefined domain sequence must be given initially. It is not suitable for GridStat due to the fact that the disjoint paths selection in GridStat depends on network resource availability. Thus it is likely that primary and alternative paths take two distinct

domain sequences.

2.4 Algorithms and Protocols

In this section, a set of relevant mechanisms, algorithms and protocols are briefly described. It lays the groundwork for the complete inter-domain routing protocol which will be presented in the following two chapters.

2.4.1 Topology Aggregation Mechanism

Due to confidentiality and scalability issues, populating all the internal link state information among all domains is infeasible. The internal topology of a domain should not be revealed to the outside. However, the lack of domains' internal topology increases the difficulty in finding an optimal or even a feasible end-to-end path. To resolve these conflicting forces, a domain must balance which information to disclose and which to hide. With topology aggregation mechanism, we can reach a point where confidentiality is maintained, meanwhile the optimal paths computation is also enabled. Moreover, by only advertising the $aggregated\ topology$ of a domain, the scalability is improved as well.

The topology of a domain can be aggregated by three means: single-node structure, star structure and full mesh structure between border nodes. We go over these three structures to find out the best solution for GridStat.

For the single-node approach, the domain is identified as a single super-node and none of the internal topology is exposed to the outside world. In the star aggregated topology, each border node connects to a virtual central node. A domain forms a star with a virtual node in the center surrounded by border nodes. In a full mesh structure, a fully connected mesh of border nodes is advertised to other domains.

For the purpose of computing the inter-domain disjoint QoS constraint path, the information provided by single-node and star-nodes structure would not be able to fulfill the needs. However with the mesh aggregated topology, we can achieve our goal. Also the mesh solution is fairly efficient considering a domain tends to be large (network nodes, edges), but number of border nodes are relatively small.

2.4.2 Hello Protocol

The hello protocol plays an important role in domain discovery when a router is initiated. For a domain D, it is used for advising its presence as well as the links and neighbors to which it is connected to the neighboring domains, and to learn the rest of network through neighbors.

There are two aspects that need to be addressed in regard to the hello protocol: building the routing table of adjacent nodes; and node discovery and related routing entry update. When a router is initiated, $hello$ messages are sent to neighbors. After receiving the ack messages, the routing pair neighbor relationship is set up. Furthermore, the $hello$ message is also periodically sent to the neighbors to ensure their availability. After pairing with neighbor nodes, a node immediately receives path vector updates from the neighbors, and performs route computation, then it sends back the updated routing information if it's necessary.

2.4.3 Path Vector Routing Protocol

The path vector routing protocol is a more recent concept compared to both the distance vector protocol and the link state protocol [2]. It is usually closely related to *Border Gateway Protocol (BGP)* [28], the *de facto* inter-domain routing protocol in Internet. BGP is a specific instance of the general path vector protocol. In this thesis, we concentrate on the general principles and mechanisms of the path vector protocol, and present a revised version in the next chapter in order

to make it suitable for the proposed peer-to-peer architecture.

The main distinction of path vector protocol from the distance vector protocol is that a node not only receives a distance vector from neighboring nodes, it also receives the entire path in regard to the destination. The information in the full path list enables easy detection of loops in the network. Therefore two route properties must be maintained in each node: destination path and next-hop.

The basic message format of Path Vector Protocol is as follows.

source	destination	cost	node list of path

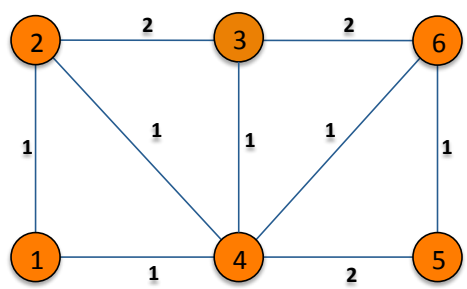

Figure 2.7: Path Vector Example

We will use Figure 2.7 as the network topology for illustration of the path vector protocol.

Consider the scenario that node 1 is receiving the path vector information from node 4. Node 4 is aware of its neighbors 2, 3, 5, 6 and needs to send the reachability information in table 2.1 to node 1.

Table 2.1: Node 4 Perspective

source	destination	cost	path
4	4	0	(4)
4	3	1	(4, 3)
4	2	1	(4, 2)
4	5	2	(4, 5)
4	6	1	(4, 6)

Before receiving the path vector information from node 4, the table of node 1 is :

Destination	Cost	Path
2	1	(1, 2)
4	1	(1, 4)

Upon receiving path vectors from node 4, the path table is updated to be:

Destination	Cost	Path
2	1	(1, 2)
3	2	(1, 4, 3)
4	1	(1, 4)
5	3	(1, 4, 5)
6	2	(1, 4, 6)

Another advanced feature which differs from distance vector is path caching allows a node to cache multiple non-looping paths to a destination from different neighbors [2]. The table below shows the cached paths from node 2 to destination 6.

Table 2.2: Table For destination 6

From Node	To Destination	Cost	Path Table Entry
2	6	3	(2, 3, 6)
2	6	3	(2, 4, 6)
2	6	3	(2, 4, 3, 6)

2.4.4 Algorithms

In the paper "Reliable Routing with QoS Guarantees for Multi-Domain" Sprintson et al. [29] proposed a polynomial-time distributed algorithm to find inter-domain link disjoint paths. The problem is known as *Min-Sum* and defined as follows: Given a graph G, a source s, a destination t, find two link-disjoint (s, t)-path P_1 and P_2 of minimal total weight $W(P_1) + W(P_2)$. The algorithm is based on Suurballe-Tangel's centralized edge disjoint algorithm [25]. For complete references, both algorithms are documented in Appendix A.

The distributed algorithm consists of 3 steps. (1) Computing the aggregated representation; (2) Computing the minimum weight of shortest paths; (3) Establishing QoS paths. It allows a source PCE to perform the inter-domain disjoint paths computation and requires $O(|B|^3)$ invocations of shortest path algorithm for each PCE in step one where $|B|$ is the number of border nodes in a domain. Based on the GridStat pub/sub paradigm, the computational complexity of computing the aggregation representation can be reduced to $O(|B|)$. The detailed analysis will be presented in Chapter 4.

In addition, Irava proposed heuristics for constructing survivable and non-survivable low-cost delay-constrained multicast graphs [22]. The goals of the heuristics are: 1) Meet end-to-end delay requirements; 2) Reduce the total cost of the multicast graph; 3) For each source destination

node pair, finding two node-disjoint paths.

The edge-priority based dynamic weight heuristic (DWH) constructs non-survivable, low-cost, delay-constrained multicast trees. The dynamic weights on edges are associated with the delay bound requirements. They are referred to as the effective-edge-costs. The dynamic edge weight serves as a relative metric of determining whether the edge is included in the paths. In addition, the dynamic weight disjoint path pairs (DW-DPP) heuristic is the survivable variant of DWH. In order to reduce the multicast tree cost, it allows sharing edges in paths. It also uses effective-edge-cost as the relative metric which is influenced by delay.

The heuristics use delay-influenced cost instead of actual cost of edge in the path computation so that it favors edges already present in the multicast tree [22]. For each new destination computation, it requires the complete multicast graph information among all domains for each publication. This requirement is impossible in inter-domain environment due to the scalability issue.

CHAPTER THREE

GRIDSTAT INTER-DOMAIN ROUTING PROTOCOL

In this chapter, the revised peer-to-peer management plane framework is presented, and the inter-domain routing module is introduced. A major role of this protocol is to exchange routing information in a consistent and standardized manner. It also enables brokers to discover and locate the others. Furthermore, it facilitates route computation in brokers.

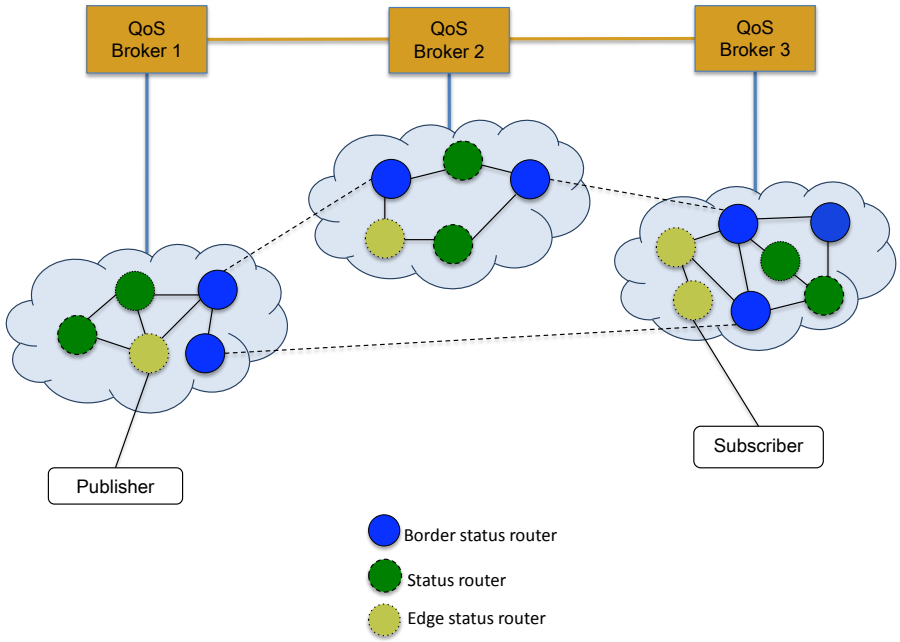

Figure 3.1: GridStat P2P Management Plane Framework

The peer-to-peer (P2P) architecture for the management plane only involves one type of broker - QoS broker. Without a hierarchical structure, several protocols must be applied in a

cooperative way for routing information exchange and inter-domain disjoint paths computation.

Because of the significant reliability requirement for routing information, a TCP-based communication session is established between adjacent QoS brokers through static configuration when they are initially activated. The session is used for brokers to exchange and update routing information and is required to stay connected. If the session is broken for some reason, the broker involved in the broken session should discard the routing information learned from the other side and re-try to establish the connection.

3.1 GridStat Inter-Domain Network Model

Consider a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of links. Each link $e \in E$ is assigned a positive delay $d(e)$ with the additive property. Required delay bound is denoted by Δ.

We use a subscript to denote a node id and use a superscript to denote a domain id. Data planes are denoted by $D^1 = (V^1, E^1), ..., D^k = (V^k, E^k)$ each of which is also referred to as a routing domain. For a domain D^i, We denote by $broker^i$ the QoS broker, B^i the set of border nodes, A^i the aggregated representation and PUB^i the publication information. In addition, D^{pub} denotes the publication domain containing the publisher whereas D^{sub} denotes subscription domain containing the subscriber. In D^{pub} (D^{sub}) the QoS broker is denoted by $broker^{pub}$ ($broker^{sub}$). The set of inter-domain link is denoted by E_{it}.

For example, a routing domain D^1 is shown in Figure 3.2 (taken from [29]). D^1 has 4 border nodes, ${b^1}_1$, ${b^1}_2$, ${b^1}_3$, ${b^1}_4$. The number shows the delay of the edges. We will use it as the domain example throughout this chapter.

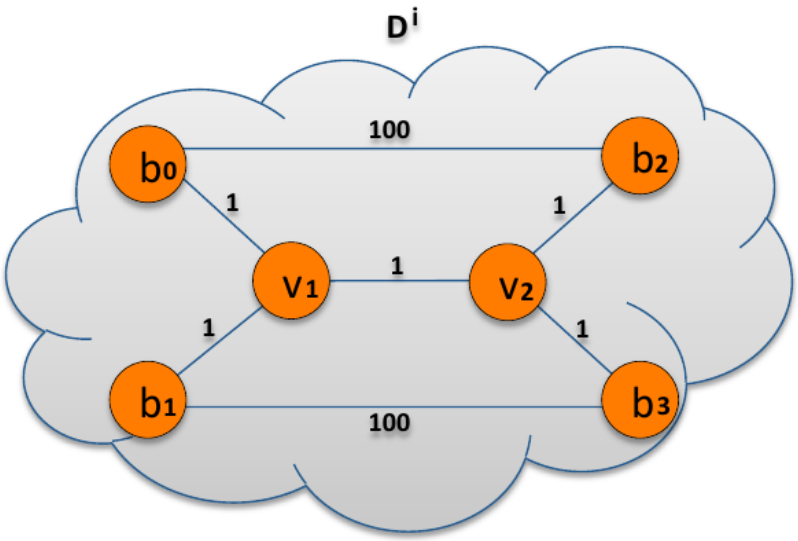

Figure 3.2: Domain Example

3.2 Inter-domain Routing Framework Overview

The inter-domain routing framework stems from a revised version of the path vector protocol (described in Chapter 2). It is also inspired by the Path Computation Element Protocol (PCEP) [17].

The revised path vector protocol is for communicating between brokers, particularly for discovering and locating, so a broker is able to exchange messages with any other broker in the management plane. After its view of the network state has converged, a broker builds a routing table to all the brokers. What differs from the typical path vector protocol is that in addition to exchange route information, a aggregation topology and publication information in the data plane for domain D^i are flooded among domains as well. It simply because without sharing aggregation topology or any kind of networking information in the data plane, brokers are incapable of

computing and establishing the disjoint paths in the data plane. So that additional flooding mechanism must be employed. It allows brokers exchange data plane link state information to find inter-domain disjoint path. To sum up, the major three types of information to be populated are : Routing information; Publication advertisement; Data plane aggregation topology.

3.3 Routing Messages

Routing messages are sent between brokers in order to enable domain/publication discovery and route computation. The routing session is established over TCP due to the reliability requirement. Four key types of messages are defined in management plane, *QoS-OPEN, QoS-KEEPALIVE, QoS-UPDATE, QoS-CLOSE*.

3.3.1 QoS-OPEN

QoS-OPEN is set up over the TCP connection between adjacent brokers. There are two main advantages of TCP for this purposes: (1) Reliability and in-order delivery (2) Security is easily achieved using Transport Layer Security (TLS). In practice, initially each broker is configured with IP addresses and port numbers for establishing TCP session. Once the session is established, each side sends *QoS-OPEN* to the other end and expecting *QoS-OPEN*.

3.3.2 QoS-KEEPALIVE

QoS-KEEPALIVE is used for routing session maintenance. Once the session is established, the *QoS-KEEPALIVE* is sent periodically between two brokers as a confirmation that the session is still alive. The maximum acceptable duration indicating end node alive is referred to as *hold time* [2]. It is determined during the initial exchange of *QoS-OPEN* message. If no *QoS-KEEPALIVE* is received in the time window of *hold time*, it indicates that the session is broken. The *QoS-*

KEEPALIVE is asymmetrical, it is not responded by the receiver. If *hold time* is set to be zero in *QoS-OPEN* message, we assume that the session is completely reliable.

3.3.3 QoS-UPDATE

QoS-UPDATE is the key message which carries the reachability of management plane brokers. Whenever a broker detects a management plane network state change, i.e, a better path is discovered to a certain broker or a new broker comes alive, it sends the *QoS-UPDATE* to the neighboring brokers. If a broker becomes aware that a neighboring broker is inactive (for example, by *QoS-KEEPALIVE* message). it generates a *withdrawn* to the rest neighboring domains to announce the offline status of the broker.

Figure 3.3 shows *QoS-UPDATE* in action. In this example the management plane has five domains. QoS^1 advises its present through *QoS-UPDATE* to the neighbor domain QoS^2. The format is $(destination, (nodelist))$. QoS^2 then sends $(QoS^1, (QoS^2, QoS^1))$ to QoS^4. Upon receiving this message, QoS^4 first tests for the existence of a loop. If one is found, it simply discard the message. Otherwise QoS^4 learns that for destination QoS^1, the path is (QoS^2, QoS^1) and it constructs the table entry for destination QoS^1. The table 3.1 is from perspective of the QoS^4 after all the domain destination has been learned.

Table 3.1: QoS4 Perspective

Destination Node	Node List of Path
QoS^1	(QoS^2, QoS^1)
QoS^2	(QoS^2)
QoS^3	(QoS^1, QoS^2, QoS^3)
QoS^5	$(QoS^2, QoS^1, QoS^3, QoS^5)$

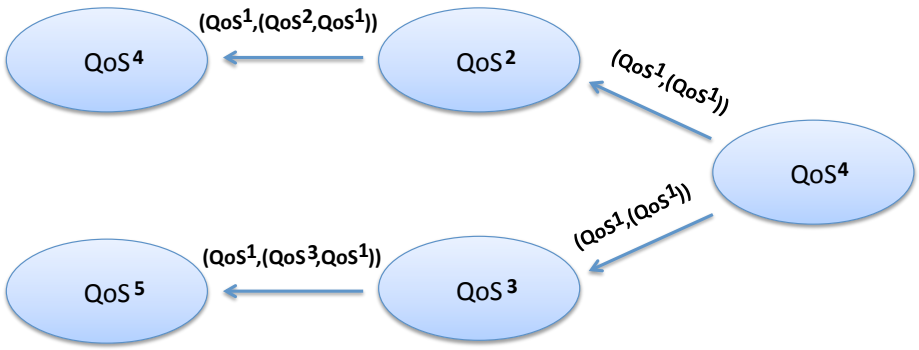

Figure 3.3: QoS Update Illustration

3.3.4 QoS-CLOSE

QoS-CLOSE is used for close a peering session between brokers. A broker may send *QoS-CLOSE* to the other end to explicitly request close connection.

3.4 Message Format

A critical functionality of path vector protocol is route advertisements. In order to facilitate route computation at each node, path vector information must be included in the routing message. Furthermore, sequence number may also be included in the routing message. It is incremented each

time when a new update is generated [2] so if a broker receives multiple route messages regarding to a same destination with different cost, it is able to distinguish which one to store and which one to discard. Therefore we have the protocol message is as shown in Table 3.2

Table 3.2: GridStat Path Vector Protocol Message Format

Destination Node	Cost	Number of Nodes	Node List of Path	Sequence Number

3.5 Broker Operations

After completion of exchanging reachability information with neighboring domains through *OoS-OPEN* and *QoS-UPDATE*, each broker then computes its domain's aggregation representation based on the data plane topology. The result of computation is the graph $A^i(V^{A^i}, E^{A^i})$, such that V^{A^i} is the set of border nodes, E^{A^i} is the set of links. For each pair of border nodes $(b^i{}_x, b^i{}_y)$, a virtual link $l \in E^{A^i}$ connects them. The link cost is obtained via any shortest path algorithm. An example A^1 of domain D^1 is shown in Figure 5.4.

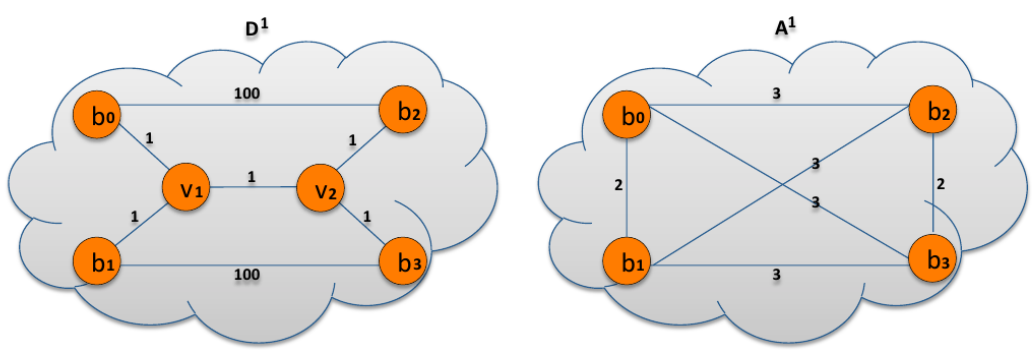

Figure 3.4: Topology Aggregation

A^i and inter-domain links E^i_{it} are then represented in a adjacency list. $broker^i$ then encodes

A_i and E^i_{it} into a *QoS-AR* message and floods it to the adjacent domains as its network topology, as shown in Figure 3.5. The format of *QoS-AR* is shown in table 3.3. As we can see from Figure 3.5 domain D^1 floods its aggregated topology A^1 and inter-domain link E^1_{it}.

Table 3.3: QoS-AR Message

domain id	A^i	L^i_{it}	sequence number

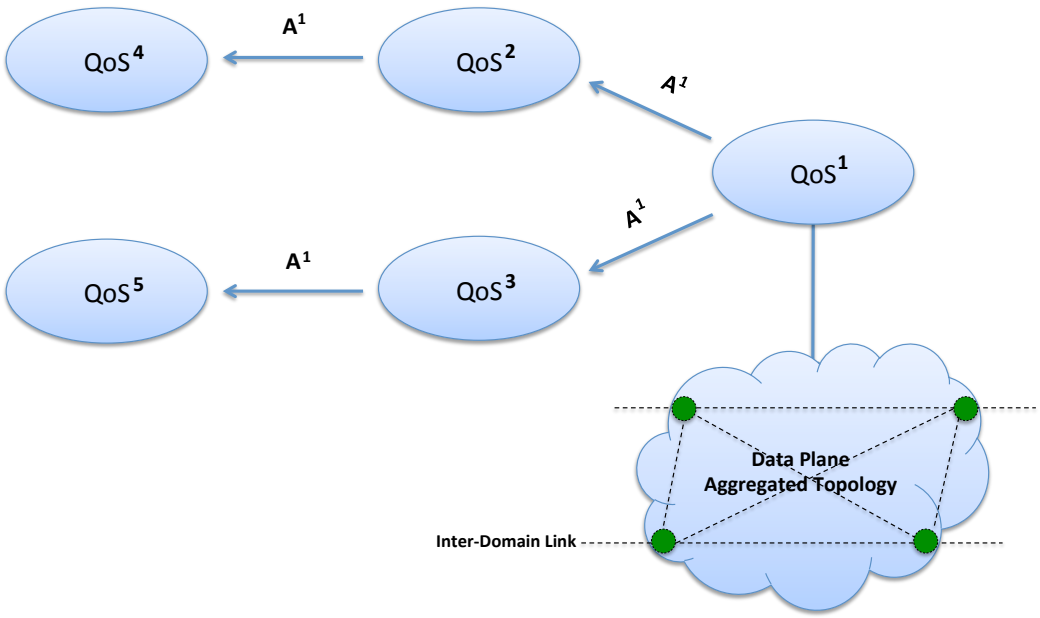

Figure 3.5: Flooding Aggregation Topology

Once $broker^i$ is aware of a publication (publisher) in domain D^i, the publication link state information PUB^i is calculated. Let ${fe^i}_{pub}$ denote the edge forwarding engine to which the publisher connects in domain D^i. The $broker^i$ constructs a virtual topology PUB^i where border

nodes are directly connected to ${fe^i}_{pub}$ by virtual links with the costs calculated via shortest path algorithm. In this case, PUB^i forms a star structure. The $broker^i$ then floods PUB^i as publication link state information of D^i. The routing message is encoded as shown in table 3.4. An example PUB^i of domain D^1 is shown in Figure 3.6.

However based on the observation that for each distinct ${fe^i}_{pub}$, there is a graph shown as Figure 3.6 to construct, so populating publication in this way may fail to scale. To solve this scalability issue, we can select a border node as the domain publication agent which is in charge of managing all the publication information in the domain. In this way only the aggregation topology is populated throughout domains and one of the border nodes is marked as the publication agent and acted as a virtual publisher for the domain. In other words, for a domain D^i, only one ${fe^i}_{pub}$ is exposed to the outside. However with only ${fe^i}_{pub}$ visible outside the domain, it is impossible to compute the end-to-end delay simply because the intra-domain delay from publication agent (a selected border node) to the actual forwarding engine is missing. Therefore for each ${fe^i}_{pub}$, we compute the shortest path from publication agent to it, and encodes all these paths delay information into a single publication message and populates it throughout domains. For simplicity, we use PUB^i as the publication information in the rest of the thesis.

Table 3.4: Publication Message Format

domain id	sequence number	time-to-live	publication	PUB^i

Upon receiving PUB from neighboring domain, $broker^i$ builds up the *pub-info* table with entry $< domain\text{–}id, publication, PUB^i >$ which tracks all active publication information.

Note that in order to establish subscription, each subscriber must locate in which domain the publication occurs. Therefore, $broker^i$ should update and flood PUB^i immediately when a new publisher announce a publication.

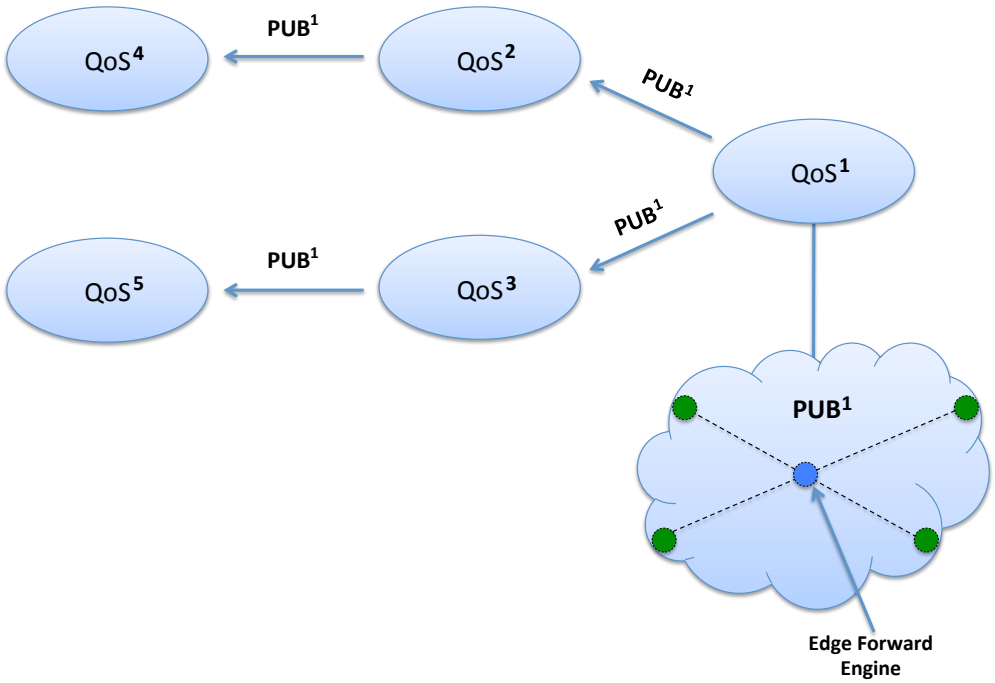

Figure 3.6: Flooding Publication information

At the end of this phase, every domain has a consistent view of *Aggregated Representation* of other domains and all publication announcement information as if it has full topology of global network.

3.6 Protocol Analysis

3.6.1 Time Complexity of Aggregation Topology Computation

The time complexity is measured by the times to invoke shortest path algorithm. For a domain $D^i = (V^i, E^i)$ with $|B^i|$ border nodes, it requires $|B^i|$ times invocations of the Dijkstra's shortest

path algorithm. If the Dijkstra's shortest path algorithm is implemented in *Fibonacci heap*, the complexity of a single invocation is $O(|V^i|lg|V^i| + |E^i|)$. Therefor the total time complexity is $O(|B^i| * (|V^i|lg|V^i| + |E^i|))$.

3.6.2 Flooding Mechanism Message Overhead

For the measurement of flooding mechanism, if the management plane $G = (V, E)$ includes V domains and E links, the total number of messages to exchange between domains is $O(|V| * |E|)$.

CHAPTER FOUR

PATH ESTABLISHMENT PROTOCOL

The path establishment protocol is a signaling protocol using source routing. Signaling protocols are an old concept in the context of circuit switching. They are used for establishing connection prior to the data transfer and maintaining the connection throughout the communication. Signaling protocols enable various entities collaborate to meet the service requirement. In GridStat, the path establishment protocol is used for inter-domain path computation and establishment. It operates in brokers in the management plane before data delivery. In other words, for a single inter-domain path, firstly it is computed and established by establishing labels on each forwarding engine in the data plane. Then the path actual carries data.

The path establishment protocol is a subscriber initiated protocol where each path computation and establishment is launched in the domain where the new requested subscription occurs. Path computation requests are handled sequentially. The computation and establishment of disjoint paths requires communication and collaboration of brokers in the domains involved.

Some types of global network information must be available to brokers in order to find the inter-domain subscription path. With the routing protocol described in the previous chapter, brokers are allowed to exchange routing and publication information among domains so that each broker is aware of aggregated global network topology. With that information in hand, each broker is able to compute the primary subscription path P_1 and initializes the subscription request via the message *MSG_PATH*. The P_1 is stored in *struct path_t* as shown in Figure 4.1. The initialed computation step occurs in the subscriber domain D^{sub} and is performed by the $broker^{sub}$.

The disjoint paths establishment procedure is described briefly as follows. $broker^{sub}$ computes P_1 and constructs the path request message MSG_PATH. It then sends the path request

```
struct path_t {
  int s;                    /* source node        */
  int t;                    /* destination  node */
  int len;                  /* path length        */
  int *pred;                /* predecessor array */
  double *dist;             /* distance array     */
  int *overlap;             /* for disjoint use   */
  int *ginfo;               /* path global info   */
};
```

Figure 4.1: $Path\_t$ Data Structure

message upstream towards D^{pub}. When a transient domain receives the message, it attaches its auxiliary topology aggregation if necessary. Consequently, the message MSG_PATH collects updated data plane link state information along the path. Upon receiving the message in the D^{pub}, $broker^{pub}$ invokes the computation of inter-domain disjoint paths.

Furthermore, another sub-protocol is integrated into path establishment protocol with role of managing multicast subscription. This sub-protocol includes multicast member join, leave, tree merge, prune, etc. Due to the fact that multicast in GridStat is in application level, the multicast mechanism can be achieved through modification of related forwarding tables in status router, in turn, the more flexibility gained compare to multicast taking place in network layer. The multicast protocol is defined by the message types described later. In the following sections, we will first examine the message types, then the complete broker operations is presented. Last but not the least, the computation complexity analysis and message over is investigated.

A multi-domain topology is shown in the Figure 4.2. Here we have a subscription domain D^{sub} , a publication domain D^{pub} and two transient domains D^1 and D^2. We will use it as an illustration example throughout this Chapter.

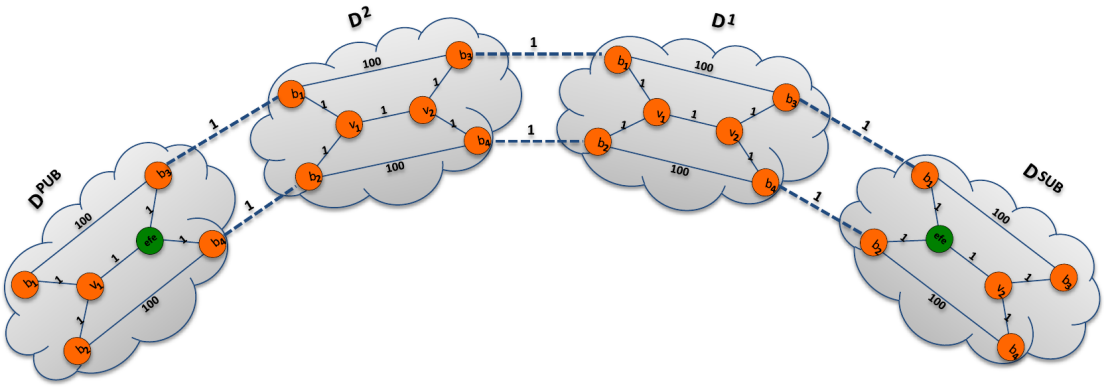

Figure 4.2: Multi-Domain Example

4.1 Broker Operations

In this section, the path establishment procedure is explained. It involves the cooperation of brokers in the management plane. It consists of three phases in chronological order. The first phase is path request, followed by path evaluation, and finally, path establishment.

4.1.1 Path Request

Let fe^{sub} be the edge forwarding engine to which the *subscriber* connects, fe^{sub} be the edge forwarding engine to which the *publisher* connects. The goal is to establish delay bounded disjoint paths between them.

Firstly, a subscriber sends the subscription request to fe^{sub} for the publication located in fe^{pub}. fe^{sub} then sends the request to $broker^{sub}$. $broker^{sub}$ then determines if the subscription takes place in the same domain, if it does, it initialize the intra-domain disjoint path computation and allocation as the current GridStat framework will do. Otherwise, $broker^{sub}$ looks up the publication table *pub-info* and try to target the publication forwarding engine fe^{pub}. Once it finds fe^{pub}, $broker^{sub}$ invokes the routine to construct the aggregated global graph G_{p_1} to compute the primary

path P_1.

G_{p_1} consists of four components, G^{pub}, G^{sub}, A^is and E_{it} and is shown in Figure 4.3.

$G^{pub} = (V, E)$, where $V = B^{pub} \cup fe^{pub}, E = \{(v, w)|v \in B^{pub}, w = fe^{pub}\}$. G^{pub} is a graph such that border nodes are directly connected to fe^{pub} by virtual links with the costs calculated via Dijkstra's shortest path algorithm. It is obtained in the publication information message via path vector protocol.

$G^{sub} = (V, E)$, where $V = B^{sub} \cup fe^{sub}, E = \{(v, w)|v \in B^{sub}, w = fe^{sub}\}$. G^{pub} is a graph such that border nodes are directly connected to fe^{sub} by virtual links with the costs calculated via Dijkstra's shortest path algorithm. G^{sub} is computed by $broker^{sub}$.

A^is is the aggregation topology of all other domains. They are obtained through the path vector protocol.

E_{it} is the set of inter-domain links.

Next the primary path P_1 is computed via shortest path algorithm with destination sets to fe^{pub} whereas source sets to fe^{sub}. Figure 4.3 illustrates the primary path P_1. P_1 is represented in terms of explicit hops of the aggregation topology level, for example (fe^{sub}, $b1$, $b3$, $b6$, $b8$, $b9$, $b11$, fe^{pub}).

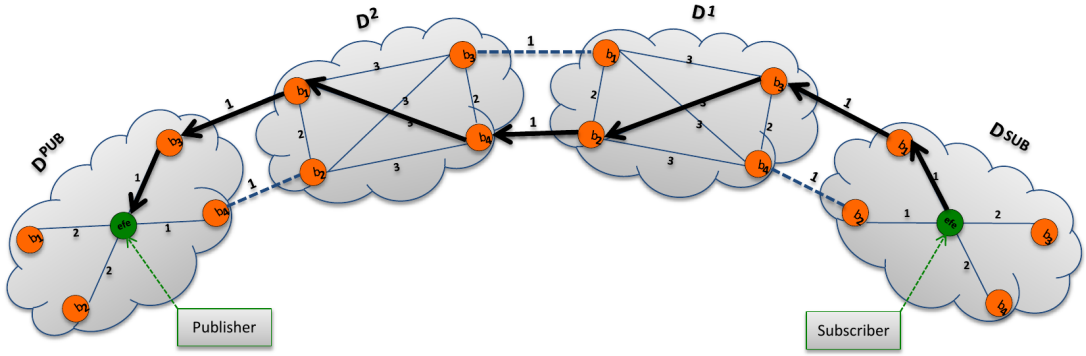

Figure 4.3: Primary Path P1

$broker^{sub}$ then constructs MSG_PATH in three steps. It first examines P_1 to find the *path_next* which indicates next domain on the P_1. In the example shown above, the path_next is the domain D^1. It is then encoded into header of the MSG_PATH message. Next it invokes *serialize_path* to encode P_1 into the message. Finally, It invokes *serialize_auxar* to include auxiliary topology aggregation of D^{sub} into the message. The computation of auxiliary topology aggregation performs the following steps.

1. Identify the path segment of the domain D^{sub} in the P_1, for example, the path portion of P_1 is $fe^{sub} - b_2$, denoted as P_1^1.

2. Reverse and negate the P_1^1 in G^{sub}, we have a graph G^{sub}_{fe,b_2}.

3. Compute the auxiliary topology array $G^{pub}_{fe,b_2} = (V, E)$, where $V = B^{sub} \cup fe^{sub}, E = \{(v, w)|v \in V, w \in V, v \neq w\}$. In particular, for any two nodes in G^{pub}_{fe,b_2}, the minimal weight path between them, calculated by modified Dijkstra's shortest path algorithm. This requires $|B| + 1$ times invocations of the algorithm. $|B|$ is the number of border nodes, 1 is the node fe^{sub}.

$broker^{sub}$ then sends the MSG_PATH to the neighbor domain identified by the path_next. Upon receiving MSG_PATH, the broker performs the following steps:

1. Parse the MSG_PATH message, obtains P_1.

2. Check whether the path_next is itself. If it is not, simply relays the message to the neighbor identified by looking up the routing table with destination to be the path_next.

3. Otherwise, identify its path portion of P_1 in terms of border nodes pair (ingress, egress).

4. If P_1 traverses the domain, attaches its own auxiliary topology array, and updates *path_next*. Otherwise relays the message.

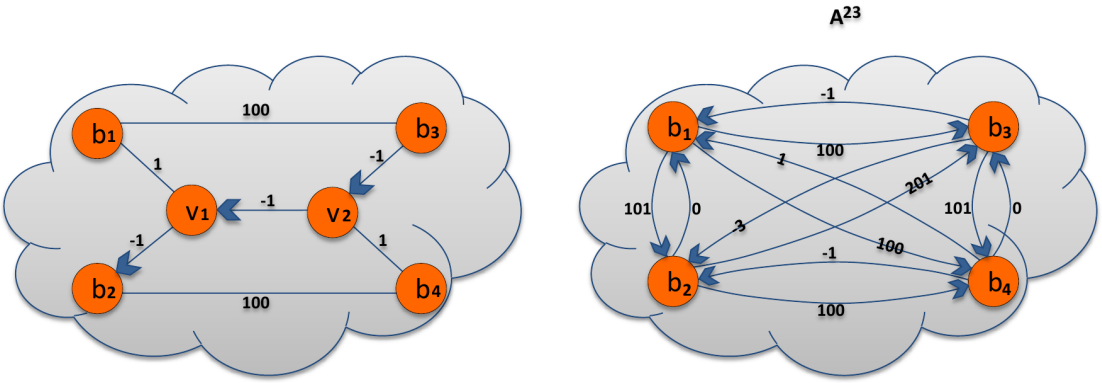

Figure 4.4: Updated Aggregated Representation $A^2_{2,3}$

In the example shown in Figure 4.3, the next domain to be traversed is D^1. When $broker^1$ receives MSG_PATH, it parses the message and obtains P_1. It then learns that the portion of P_1 in D^1 is (b_3, b_2). Figure 4.4 illustrates the transformation and computation of auxiliary topology aggregation $A^1_{3,6}$. With ingress node b_3 and egress node b_6, on the left side of Figure 4.4, we have a graph which negates and reverses the path P^1_1, denoted by $G^1_{3,6}$. Based on $G^1_{3,6}$, on the right hand, we have the auxiliary aggregation topology $A^1_{3,6}$ in which for each border nodes b_i and b_j, a path computed between them computed by modified Dijkstra's shortest path algorithm.

Figure 4.5 illustrates the auxiliary aggregated topology in D^2. The ingress node is b_4 and the egress node is b_1, the $A^2_{4,1}$ is then computed on the right hand side.

Upon receiving MSG_PATH at D^{pub} (publication domain), $broker^{pub}$ can compute the secondary link disjoint path using: (1) Reversed and negated aggregated link information for each domain on the path, acquired from the MSG_PATH message. (2) Inter-domain link information E_{it}, acquired from the path vector protocol. Figure 4.6 illustrates the computation result of the

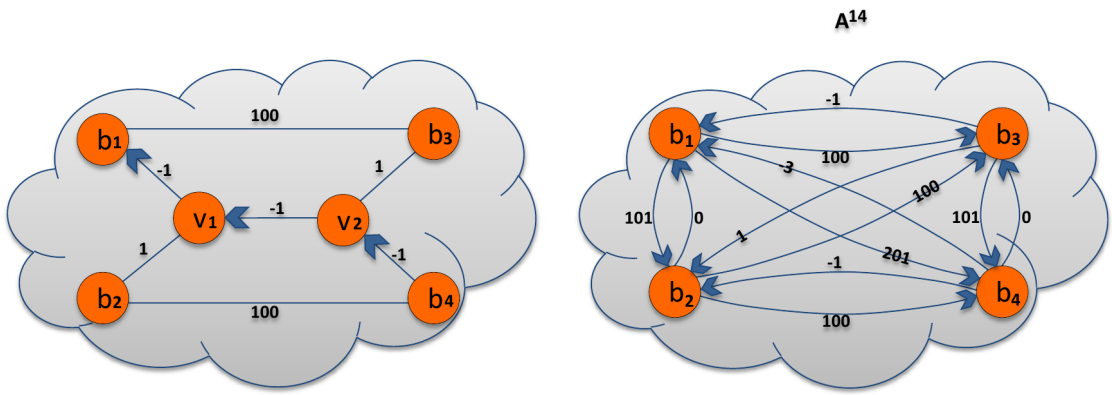

Figure 4.5: Updated Aggregated Representation $A^1_{1,4}$

disjoint paths at the domain aggregation representation level (sub set of border nodes).

However, $broker^{pub}$ does not know the intra-domain paths so it is unable to evaluate the exact delay of $d(P_1)$ and $d(P_2)$ at this point, so it cannot verify whether the computed path satisfy the delay bound.

For the sake of evaluating the primary path P_1 and backup P_2, the evaluation phase is required and presented below.

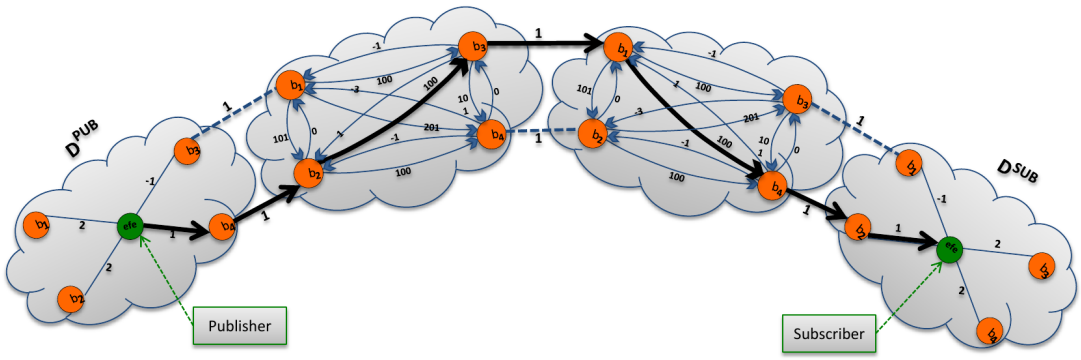

Figure 4.6: The Secondary Path Computation

The complete path request procedure for brokers are illustrated in Figure 4.7.

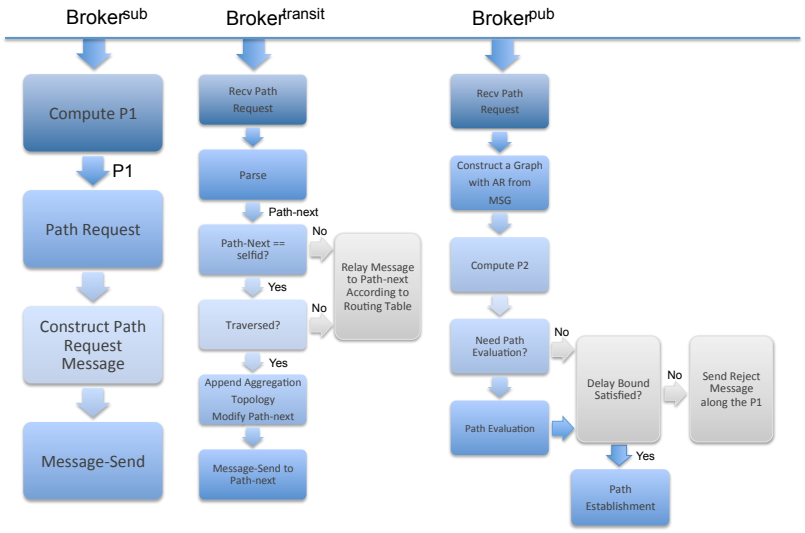

Figure 4.7: Path Request Brokers Operations

4.1.2 Path Evaluation

$broker^{pub}$ performs the following steps to evaluate delay of P_1 and P_2. Recall from Chapter Two, Δ denotes bounded delay.

1. If $d(P_1) <= \Delta \cap d(P_2) <= \Delta$, processes to path establishment phase.

2. If $d(P_1) + d(P_2) > 2 * \Delta$, it means that at least one of the two paths violates the delay bound, denies request and sends the reject message back to P_1 and P_2.

3. If $d(P_1) + d(P_2) <= 2 * \Delta$ and one of the two paths violates delay bound: $(d(P_1) > \Delta$ or $d(P_2) > \Delta)$, there are two cases to handle:

   1. If P_1, P_2 overlaps in a domain D^i, sends MSG_EVAL to $broker^i$ and wait for the evaluation result MSG_EVAL_ACK. Upon receiving the evaluation result message, $broker^i$ evaluates $d(P_1)$ and $d(P_2)$, if one of the paths violates the delay bound, it denies request and sends the reject message back to P_1 and P_2. Otherwise goes to path establishment phase.

2. If P_1, P_2 do not overlaps in any domain, it means one of the two paths violates the delay bound. It then denies request and sends the reject message back to P_1 and P_2.

Because the message MSG_EVAL only sends to the domain where P_1 and P_2 both traverse, the format of message is defined as follows:

domain id	publication	$P1(ingress, egress)$	$P2(ingress, egress)$

When D^i receives the *MSG-EVAL*, it invokes the path convergence routine to compute the actual intra-domain paths and more importantly, the corresponding delay. Then the information is encoded in a *MSG-EVAL-ACK* and sends back to $broker^{pub}$. The format of the message is shown as follows:

publication	$d(P1)$	$d(P2)$

4.1.3 Path Allocation

Path allocation is the last phase of disjoint path establishment. After evaluating the delay bound of each path, two aspects affect the construction of the path acknowledge message, *MSG_RSVP*: (1) If both P_1 and P_2 traverse the same domain D^i, the two sub-paths are encoded into a single message; (2) If either P_1 or P_2 traverses a domain D^i, the sub-path is encoded into the message.

$broker^{pub}$ sends the MSG_RSVP message backward along the primary path P_1 and the backup path P_2. The disjoint path pair is established by the cooperation between brokers, each of which constructs segments of the paths across one domain. Upon receiving the MSG_RSVP message, brokers along the paths then build up the forwarding table for data delivery. Two cases may occur:

1. Domain D^i is traversed by path P_1 or P_2. $broker^i$ will receive the message MSG_RSVP designating the intra-domain path, it simply builds up the forwarding table for data delivery.

2. Domain D^i is traversed by both path P_1 and P_2. Since $broker^{pub}$ sends the message MSG_RSVP on P_1 and P_2, $broker^i$ will receive two identical messages. It simply discards the latter one, and performs the path convergence routine to establish the two disjoint intra-domain paths by setting up the forwarding table in the data plane.

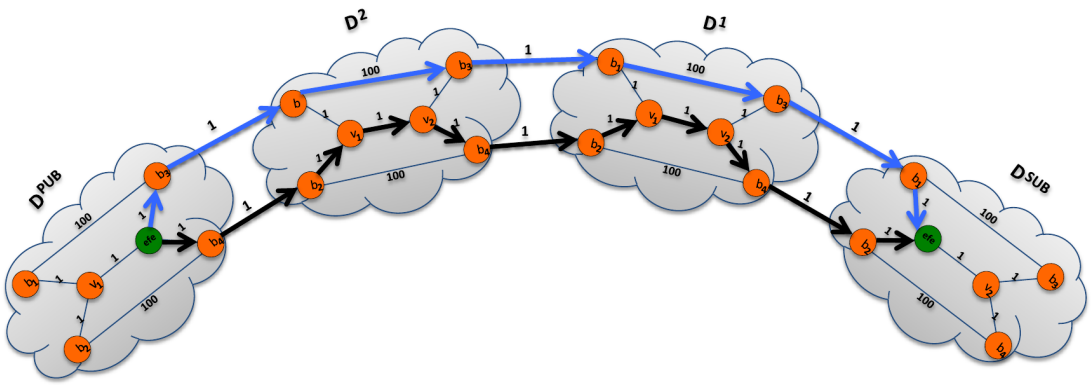

Figure 4.8: Two Disjoint Path Computed

At this point, we have the forwarding tables have been built in the data plane, as shown in Figure 4.8.

The complete broker operations for path evaluation and path allocation are illustrated in Figure 4.9. On the left hand side of Figure 4.9 path evaluation phase is presented, while on the right hand, path allocation phase is illustrated.

4.2 Message Types

Basically the messages in path establishment protocol falls into three sub-categories, path establishment, path evaluation and path teardown. Path establishment messages are sent between do-

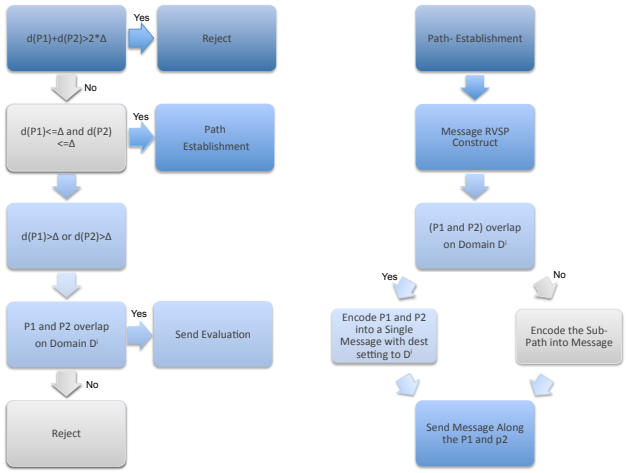

Figure 4.9: Path Evaluation and Allocation Broker Operations

mains along the path and include MSG_PATH and MSG_RSVP types of messages. Path evaluation only involve domains that path P_1 and P_2 both traverse and include MSG_EVAL, MSG_EVAL_ACK types of messages. Moreover, publication terminations are specified by MSG_PUB_TEAR messages, while MSG_SUB_TEAR is used for signaling the subscription termination. We will look into each of the messages in the following section.

4.2.1 MSG_PATH

MSG_PATH is issued by the broker in which the subscriber resides. The message plays an important role of enabling disjoint path computation by collecting auxiliary aggregated topology information along the path. When a domain receives MSG_PATH, the updated aggregated topology is computed and appended in the message. The key element in MSG_PATH is the primary path P_1. The message format is shown as table 4.1

Table 4.1: MSG_PATH

msg-header	next-stop	P_1	ARs

4.2.2 MSG_RSVP

MSG_RSVP is path acknowledge message sent from publisher domain. If both paths P_1 and P_2 traverse a domain D^i, both paths will be included in the message. If only one of two disjoint paths traverses a domain D^i, only the path cross the domain is encoded in the message. The message format is shown as table 4.2

Table 4.2: MSG_RSVP

Primary path P_1	Backup path P_2

4.2.3 MSG_EVAL

MSG_EVAL message is issued from publication domain to domains in which the evaluation process is required. It plays an important role of acquiring the necessary intra-domain paths delay information for computing the global end-to-end delay bounded paths. The process is discussed in the previous section. The message format is shown as table 4.3.

Table 4.3: MSG_EVAL

source domain	destination domain	first border nodes pair	second border node pair

4.2.4 MSG_EVAL_ACK

Upon receiving MSG_EVAL messages, a broker invokes intra-domain path convergence function to evaluate sub-path cost of P_1 and P_2 respectively. Next the intra-domain path is sent back to the publisher domain. With the information obtained, the delay bound of both paths can be computed, which is needed in constructing the MSG_RSVP message. The message format is shown as table 4.4.

Table 4.4: MSG_EVAL_ACK

source domain	destination domain	primary sub-path delay	backup sub-path delay

4.2.5 MSG_PUB_TEAR

MSG_PUB_TEAR is issued by publication domain. As its name suggested, it indicates the termination of the publisher. Basically it will remove all the forwarding states regarding this publication in the data plane. Upon receiving a MSG_PUB_TEAR message, a broker removes the intra-domain forwarding entries and relays the message to the downstream domains. The message format is shown as table 4.5.

Table 4.5: MSG_PUB_TEAR

pub domain	publication

4.2.6 MSG_SUB_TEAR

MSG_SUB_TEAR is issued by the subscription domain. It removes the local forwarding entries, then it checks if there exists any other subscriptions. If it is the only subscription in the domain, the message is sent upstream toward the publication domain. If it arrives at the publication domain, it implies that there are no subscriptions. Otherwise the message will be stopped at the point where branch subscription exist. The message format is shown as table 4.3.

Table 4.6: MSG_SUB_TEAR

sub domain	publication	explicit subscription path

4.3 Protocol Analysis

4.3.1 Message Overhead

The analysis is based on a n domain management plane setup. In the worst case, a path request message traverses n domains, each domain appends its updated AR, the message size of AR is $|B|^2$, so the maximum message size is $O(n * |B|^2)$ In the average case, a path request message traverses $n/2$ domains, each domain appends its updated AR, the size of AR is $|B|^2$, so the average message size is $O(n * |B|^2/2)$

4.3.2 Computation Analysis

To establish a pair of disjoint paths, it requires a subset of brokers to operate in a collaborative fashion. A domain is denoted as a graph $G = (V, E)$ with $|B|$ border nodes. The decomposed analysis is presented.

- Computation in D^{sub} The primary path P_1 is computed over global aggregation topology $G_{p_1} = (V_{G_{p_1}}, E_{G_{p_1}})$ where $V_{G_{p_1}}$ is the set of border nodes, $E_{G_{P_1}}$ is the aggregation topology links and inter-domain links. The computation procedure is analyzed as follows:
  - 1 invocation of Dijkstra's algorithm over G_{p_1}, as we mentioned in previous chapter, the time complexity is $O(|V_{G_{p_1}}|lg|V_{G_{p_1}}|+|E_{G_{P_1}}|)$ with implementation of *Fibonacci heap*.
  - Computation of auxiliary AR, it requires $|B^{sub}|$ times to invoke Dijkstra's shortest path algorithm, the time complexity is $O(|B^{sub}| * (|V^{sub}|lg|V^{sub}| + |E^{sub}|))$.
- Computation in the transit domain D^i

  The operation in transit domains only involves the computation of auxiliary AR. It requires

$|B^i|$ times to invoke Dijkstra's shortest path algorithm, the time complexity is $O(|B^i| * (|V^i|lg|V^i| + |E^i|))$.

- Computation in D^{pub} The operation in publication domain involves construction of auxiliary global aggregation topology. With the aggregation topology inquired from path establishment message, the auxiliary global AR $G_{p_2}(V_{G_{p_2}}, E_{G_{p_2}})$is constructed. The graph size is identical with G_{p_1}. Then one Dijkstra's modified shortest path algorithm is invoked. The complexity is $O(|V_{G_{p_2}}|lg|V_{G_{p_2}}| + |E_{G_{p_2}}|)$.

Therefore the total time complexity to compute the inter-domain disjoint path is

$$O(|V_{G_{p_1}}|lg|V_{G_{p_1}}|+|E_{G_{P_1}}|)+\sum_{i=1}^{n} O(|B^i|*(|V^i|lg|V^i|+|E^i|))+O(|V_{G_{p_2}}|lg|V_{G_{p_2}}|+|E_{G_{p_2}}|)$$

where n is the number of domains traversed by P_1.

CHAPTER FIVE

PROTOTYPE IMPLEMENTATION AND EVALUATION

The protocols and algorithms of the previous Chapters have been implemented as an inter-domain QoS routing module prototype for GridStat. It is a fully functional routing module which is implemented in C with 2653 lines of code. For simplicity and demonstration, the current implementation uses the UDP connections. It can be easily replaced by the TCP connections for reliable routing communication. The chapter is organized as follows. Section 5.1 describes the basic data structure. Section 5.2 describes the messaging interface. Section 5.3 describes the routing state machine. Section 5.4 describes the experiment setup. Section 5.7 describes the path vector protocol evaluation. Finally, section 5.8 describes the path establishment protocol evaluation.

5.1 Data Structure

```
struct domain_t {
  int id;                        /* domain id */
  char ip_addr[16];              /* router ip address */
  int portnum;                   /* router port */
  int send_flag;                 /* send flag */
  int nr_neigh    ;              /* number of neighbors */
  neighbor_t *neigh_lst;         /* information for neighbors */
  path_vector_t **path_table;    /* path vector table */
  graph_t dp;                    /* data plane representation */
  graph_t global;                /* global graph */
  int8_t *recv_ar;               /* received aggregation flag*/
  int pvupdate;                  /* PV update count */
  int arupdate;                  /* AR update count */
};
```

Figure 5.1: $Domain\_t$ Routing Data Structure

Domain_t is the core data structure for the routing module. It is maintained by QoS brokers

for domain discovery and path establishment. It is listed in Figure 5.1 with comments describing each field.

Path_table_t is used for storing path vector information learned from adjacent domains. For each destination domain *dest*, it can be configured in two modes, path caching enabled or disabled. If a broker runs in the *cached mode*, it maintains a set of paths for domain *destid* in ascending order in a linked list. On the other hand, if it runs in *un-cached mode*, the *next* field is *NULL* and only the shortest path is stored. To achieve better caching performance, a priority heap data structure could be adopted which results in *O(log N)* insertion complexity and *O(1)* to return a minimal cost path. The *path vector* structure is listed in Figure 5.2 with comments describing the entries.

```
struct path_vector_t {
 int destid;                    /* destination domain */
 struct pv {
   int cost;                  /* path cost */
   int next_hop;              /* next hop (neighbor id) */
   int *path;                 /* path array */
   struct pv *next;           /* link list for caching */
 } pv;
};
```

Figure 5.2: $Path\_vector\_t$ Routing Data Structure

Figure 5.3 is the graph representation of the data plane. The graph is represented in the format of adjacency list *adj*. *AR* is the aggregation representation of the graph. It is precomputed during the initialization phase and the data structure is listed in Figure 5.4. For each border nodes pair, a shortest path *pri* is computed, computing it uses Dijkstra's shortest path algorithm $|B|$ times. Meanwhile, a corresponding auxiliary array paths are also computed, it requires $|B|^2$ times invocation of modified Dijkstra's shortest path algorithm. The routine to compute auxiliary aggregation topology is listed in Figure 5.5. From routine we know for a primary path pri, the routine negates

and reverses it, then computes the auxiliary graph and stores it to bks.

```
struct graph_t {
  int V;                /* number of vertex */
  int E;                /* number of edge */
  link *adj;            /* adjacent list */

  int B;                /* number of border node */
  int *bn;              /* border idx array */
  int *nb;              /* border position auxiliary array */
  ar_t **AR;            /* aggregation representation */

  inter_t inter;        /* interdomain link representation */
  int *top_agg;         /* topology aggregation */
  struct g_node_t {
    int domain_id;
    int node_id;
  } *global_array;        /* global array */
  int gid;              /* global node id */
};
```

Figure 5.3: $Graph\_t$ Routing Data Structure

```
struct ar_t {
  path_t pri;           /* primary path */
  path_t **bks;         /* auxiliary array for primary path */
};
```

Figure 5.4: Aggregation Topology Data Structure

Code 5.6 is the *global path segment* data structure. It is used for brokers managing and storing the path segment information of a subscription. When a broker constructs or receives a *MSG_PATH* message or a *MSG_RSVP* message, the path segment information is computed and filled in a *gps_t* data structure.

```
Graph compute_backup_ar(Graph G, path_t pri)
{
  int i, j, v, w, s, t;
  Graph G_clone = graph_clone(G);
  s = G->nb[pri->s];
  t = G->nb[pri->t];

  G_clone = graph_reverse_negate(G_clone, pri);

  for (i = 0; i < G->B; i++) {
    v = G->bn[i];
    for (j = 0; j < G->B; j++) {
      w = G->bn[j];
      if (v == w) continue;

      G->AR[s][t]->bks[i][j] =
                    graph_compute_sp(G_clone, v, w);
    }
  }

  graph_free(G_clone);
  return G;
}
```

Figure 5.5: Auxiliary Aggregation Topology Computation

5.2 Messaging Interface

Routing messages are sent and received between QoS brokers. Each message contains a common header, starts with *source domain* followed by *msg type* and then *message length*. The length of header is 8 bytes. The header format is listed in Appendix B.

The core message types allowing disjoint paths computation are listed as below. *MSG_PV* is the path vector message with the role of delivering domain reachability information to neighbor domains. *MSG_AR* contains domain aggregation topology and *MSG_PUB* conveys the publication information, both of which are populated through flooding mechanism described in Chapter 3. *MSG_PV_ACK* and *MSG_AR_ACK* are for acknowledging to neighbors of the receipt of

```
struct gps_t {
  pub_t pubinfo;            /* publication info */
  path_t p;                 /* primary path */
  int innode;               /* ingress node id */
  int enode;                /* egress node id */
  int next_hop;             /* next_hop id */
  void *msg;                /* msg in binary */
  int msglen;               /* msg length */
  int *path;                /* path array */
  int nnodes;               /* number of nodes */
  struct timeval reqtval;   /* subscription request time */
  int aroffset;             /* AR offset in msg */
  short msg_type;           /* msg type, path or rsvp */
};
```

Figure 5.6: Global Path Segment Structure

the *MSG_PV* and *MSG_AR* messages. *MSG_PATH* and *MSG_RSVP* are used for path disjoint paths establishment protocol. *MSG_EVAL* and *MSG_EVAL_ACK* are used for path evaluation use. *MSG_PUB_TEAR* and *MSG_SUB_TEAR* are used for publication tear-down messages and subscription tear-down messages.

```
#define MSG_PV          0x1
#define MSG_AR          (MSG_PV << 1)
#define MSG_PUB         (MSG_PV << 2)
#define MSG_PV_ACK      (MSG_PV << 3)
#define MSG_AR_ACK      (MSG_PV << 4)
#define MSG_PATH        (MSG_PV << 5)
#define MSG_RESV        (MSG_PV << 6)
#define MSG_EVAL        (MSG_PV << 7)
#define MSG_EVAL_ACK    (MSG_PV << 8)
#define MSG_PUB_TEAR    (MSG_PV << 9)
#define MSG_SUB_TEAR    (MSG_PV << 10)
```

Figure 5.7: Message Type

As we can see from Figure 5.7, each message type is specified by 1 bit. In this way, various types of messages can be easily combined into a single one through *binary or* operation. For

example, the following call of *msg_send* assembles *MSG_AR* and *MSG_PUB* and send it to the domain *next_hop*.

```
msg_send(node_id, next_hop, MSG_PATH | MSG_PUB);
```

5.3 Routing State Machine

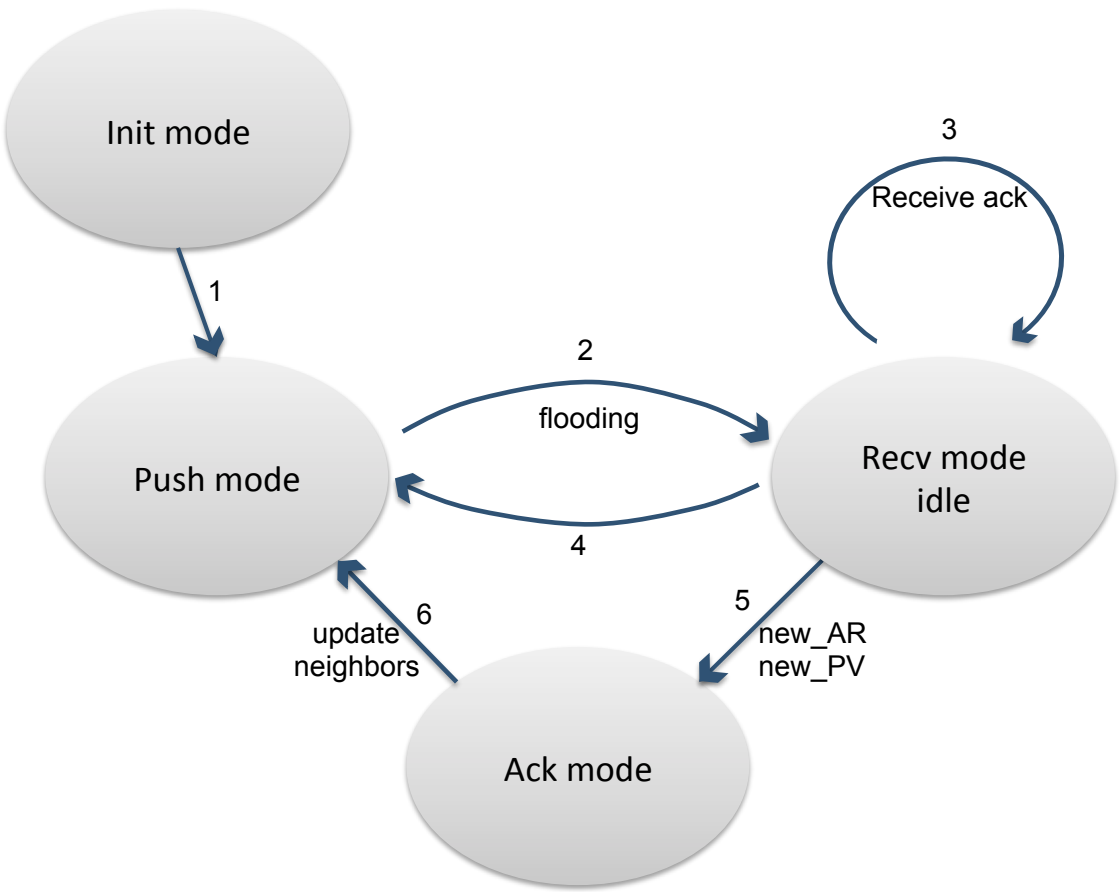

Figure 5.8: Routing State Machine

Figure 5.8 shows the core routing state machine. We will describe each transition according on the number depicted on the figure. The complete routing state transitions is listed in *code 6*.

1. Each broker is activated in the ENTRY_STATE which is *init_mode*. Messages *MSG_PV*, *MSG_AR* and *MSG_PUB* are constructed and the message destinations are set to all of its adjacent nodes. It then switches to *push_mode*

2. The broker floods the messages constructed in *init_mode* to all of its neighbors, meanwhile setting up the *hold down* timer. It then goes to *recv_mode*.

3. Upon receiving *MSG_PV_ACK* or *MSG_AR_ACK*, a broker sets the ack flag and loops back to *recv_mode*

4. When the *hold down* timer expires, the broker goes to send mode, re-sending the un-acked message.

5. Upon receiving new AR or PV messages, the broker switches to ack mode, sends ack back.

6. It then goes to *push_mode* and sends the updated routing information to the rest of neighbors.

```
#define ENTRY_STATE push_mode
#define EXIT_STATE end

struct transition state_transitions[] = {
        {push_mode, ok, recv_mode},
        {push_mode, fail, end},
        {recv_mode, ack, recv_mode},
        {recv_mode, repeat, recv_mode},
        {recv_mode, update, ack_mode},
        {ack_mode,  update, send_mode},
        {send_mode, ok, recv_mode},
        {recv_mode, fail, end},
        {send_mode, fail, end}};
```

Figure 5.9: Routing Transition Table

5.4 Experiment Testbed Setup

In order to simulate on the real network environment, we chose *Emulab* as the experiment testbed. *Emulab* is a network testbed, giving researchers a wide range of environments in which to develop, debug, and evaluate their systems . The hardware configuration is listed as below.

- Intel(R) Xeon(TM) 2-core CPU 3.00GHz
- Linux Fedora-15-64-STD with kernel 3.8.0-22-generic
- Gcc version 4.7.3
- 100 Mbit link

5.5 Topology Setup

5.5.1 Management Plane Topology

In this experiment we have set up seven topologies of the management plane to evaluate. Each node represents a routing domain, The number of domains in our test environment ranges from four to ten. QoS brokers are configured on each node running the inter-domain routing module. Figure 5.10 illustrates all the management plane topology setup in our experiment. All the edges have the same weight of 1.

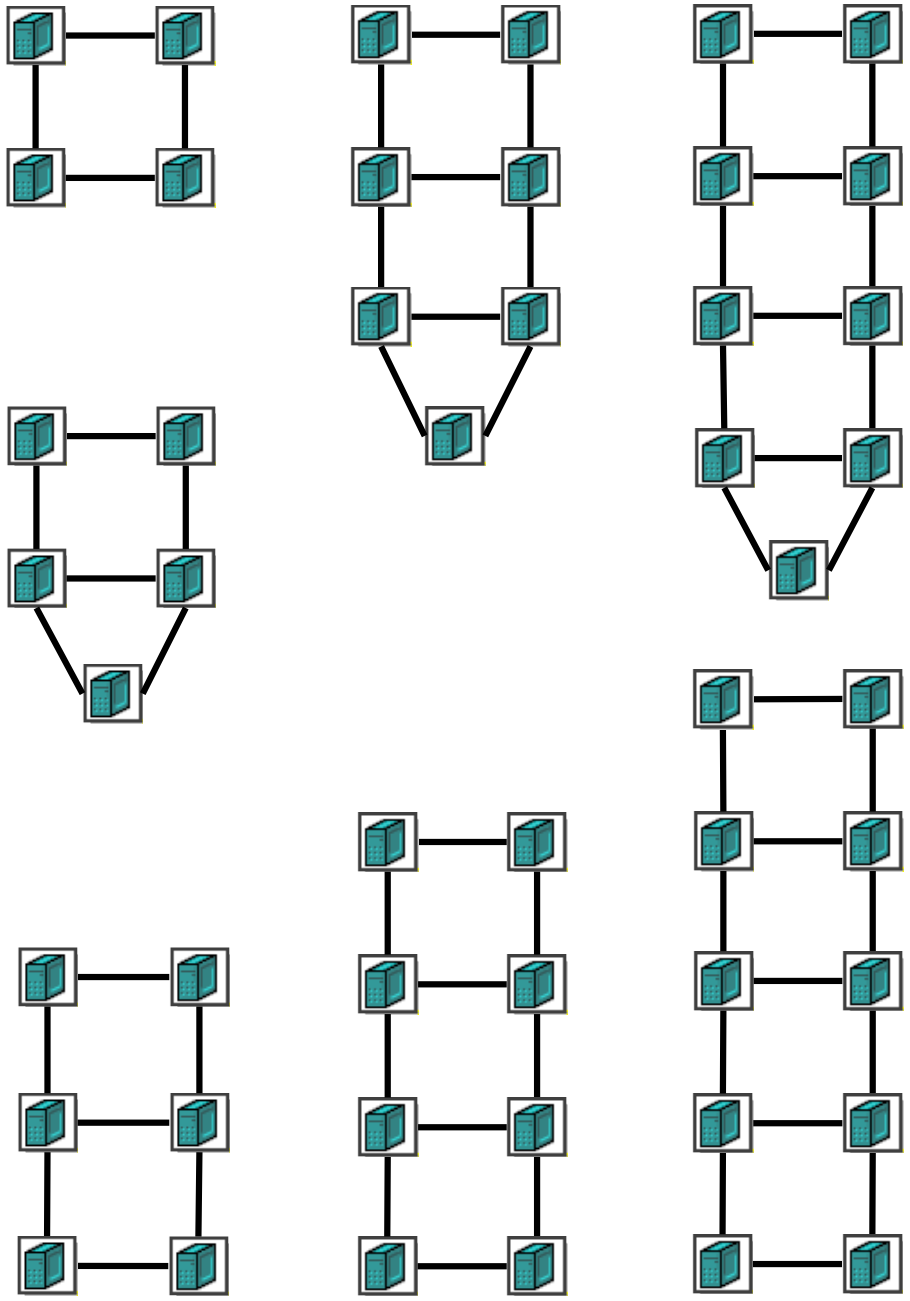

Figure 5.10: Emulab Topology

5.5.2 Data Plane Topology

Each domain contains the same topology as shown in Figure 5.11 given by Bhandari. It consists of 10 forwarding engines. We chose a subset of forwarding engines as the border nodes which connect with neighboring domains. Without loss of generality, each border node in our model is also considered as a edge forwarding engine allowing publishers or subscribers to connect to it for publishing or subscribing status variables.

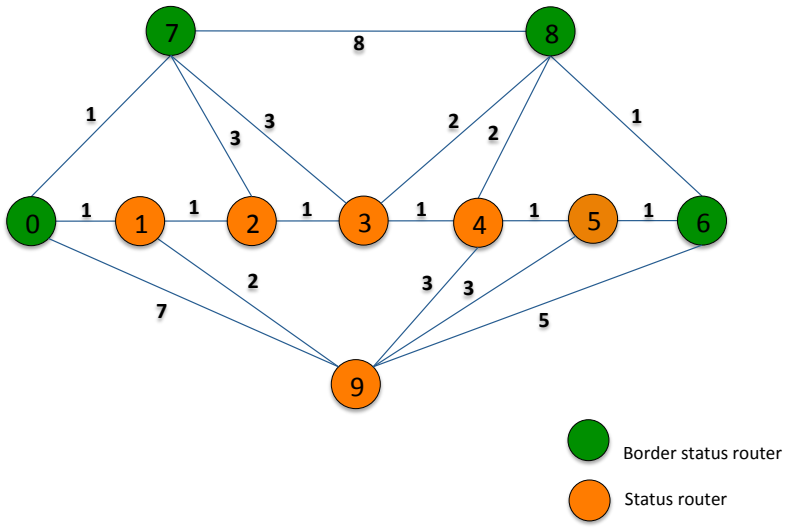

Figure 5.11: Data Plane Topology

5.5.3 Inter-domain Link

Although the management plane and the data plane are two separate networks, the inter-domain links are constructed based on the management plane topology throughout the experiments. For example we have a complete setup as shown in Figure 5.12, each data plane network peers with its neighbor via 2 inter-domain link with delay setting to 1.

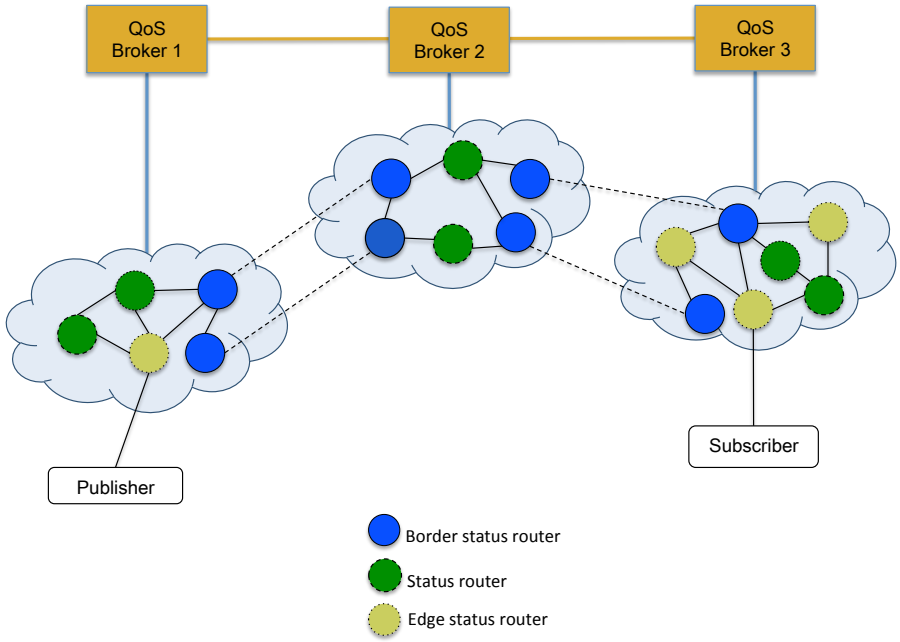

Figure 5.12: Inter-domain Link

5.6 Experiment Design

The purpose of our experiment is to evaluate the path vector protocol described in Chapter 3 and the path establishment protocol described in Chapter 4.

5.6.1 Path Vector Protocol

For the path vector protocol, the experiment mainly focus on evaluating the *convergence time* which is an important factor due to the fact that at that point routers reach a consistent view of a network state. The convergence time is evaluated according to two aspects: 1) Number of routing information updates required to converge. 2) Number of message sent prior to convergence. The reason that convergence time metric favors update and message counts rather than actual latency is that a count is a relative measurement, it is irrelevant with the CPU speed and the network environment. So it is a intricate metric for the routing protocol.

The experimental process for the path vector protocol is conducted as follows. *Total_nodes* number of the *Emulab* machines were controlled through the ssh sessions. Each machine was configured with the awareness of its adjacent neighbors. Then we started the broker routing module on each machine simultaneously. According to the metrics we defined in the following sections, the data has been gathered and analyzed.

5.6.2 Path Establishment Protocol

For the path establishment protocol, we concentrate on three factors: 1) Disjoint paths establishment delay, defined as the time difference between the moment when the subscriber receives the reply message *MSG_RSVP* from the publisher and the moment when it sends the path request message *MSG_PATH*. 2) Inter-domain disjoint paths cost in terms of delay, measured by the delay of the primary path $d(P_1)$ and delay of the back up path $d(P_2)$ respectively. Since the distributed

algorithm is based on the aggregation topology, in order to verify $d(P_1)$ and $d(P_2)$ and obtain the total minimal cost, we have also implemented a centralized algorithm, which runs over full detail. Both of the paths cost then are compared. 3) In order to prove the new capability of the fault tolerance for the management plane, we perform experiments which demonstrates the system resilience feature of the inter-domain routing module. It will be presented in Subsection 5.8.3.

The experiment process for the path establishment protocol is described as follows. A timer has been set up for automatically generating subscription requests. The timer periodically generated a SIGALRM signal. Each time the timer expires, the SIGALRM handler is invoked which performs the disjoint path establishment routine [31]. Currently we set the timer interval to be 1 second.

5.7 Path Vector Protocol Evaluation

5.7.1 Convergence Time

We mainly focus on two types of messages, the path vector message *MSG_PV* and the aggregation message *MSG_AR*. Path vector routing information is exchanged between adjacent nodes while aggregation topology is sent throughout domains via flooding mechanism.

The prototype implementation employs several simple but efficient techniques for the quick convergence. For domains aggregation topology convergence, each broker maintains a flag array $recv\_ar$ indicating the received ARs. For example, if D^i receives aggregation topology AR^j from the neighbor domain k, it checks whether the *sequence number* of AR^j greater than the current sequence number stored. If it does, updates $recv\_ar[j]$ to newly received AR^j, and set flag indicating received AR^j from the neighbor k, so $broker^i$ will not include the AR^j in the message *MSG_AR* sending to the neighbor k in the next round. Since the data plane topology is

fairly static, it tends to remain unchanged throughout our experiment. Thus the converge time for ARs is defined as the point when it receives all ARs. It also indicates that the requirement of $total\_nodes - 1$ times updates to converge.

For the path vector convergence, each domain exchanges its best path vector information with adjacent domains. A domain maintains its *local optimal* path vector to all destinations. The convergence time for measuring the path vector message is difficult than the AR message, because the status of path vector information stored in a broker is not a boolean value. For example, node 1 learns from its neighbor 2 that to destination 6, the path is (2, 3, 6) and with the cost 5. Then it updates the routing table sets next_hop to 2 for destination 6. However, node 1 may soon receive from its neighbor 4 that the path (2, 4, 6) and the cost is 3, then node 1 updates the corresponding routing path table entry. One may say the node 1 has been converged at this point. However the node 1 may learn some other better paths in the near future. Therefore to measure the convergence time of path vector messages, a timer is necessary. When an updated path vector message is received, simply re-initializes the timer. The expiration of the timer indicates during that time period, there is no new update in routing table. We claim that the path table is converged at that point.

Figure 5.13 illustrates the average updates count of *MSG_PV* per node due to convergence. Whenever a broker receives a better path through path vector protocol, it increments the update counter *pvupdate* defined in the *domain_t* structure. Due to converge, we add the *pvupdate* of each node and get the average value. The expected count of path vector updates is a function of topology, edge cost and update order. We take a four node topology as an example. From node 1's point of view, if all the edge cost are set to 1, it will only need 1 update to reach convergence for node 1.

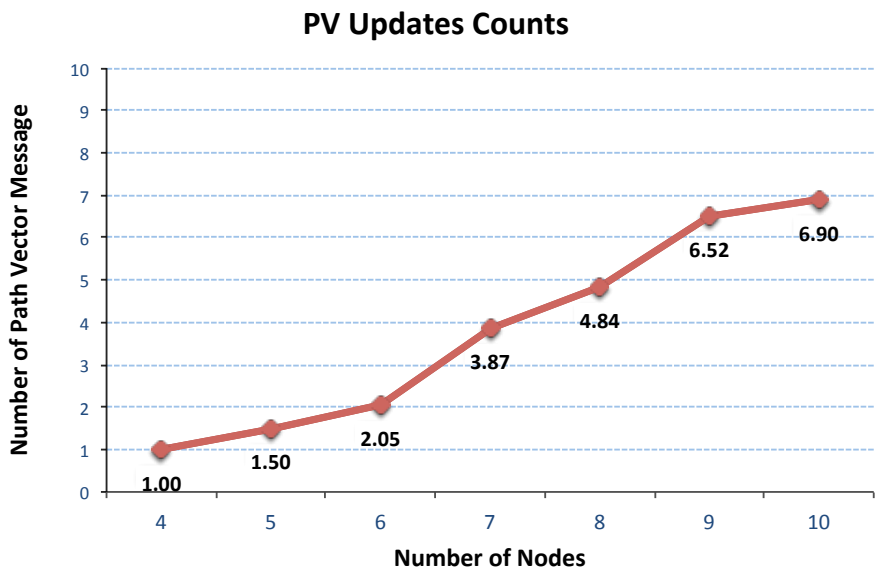

Figure 5.13: Path Vector Updates

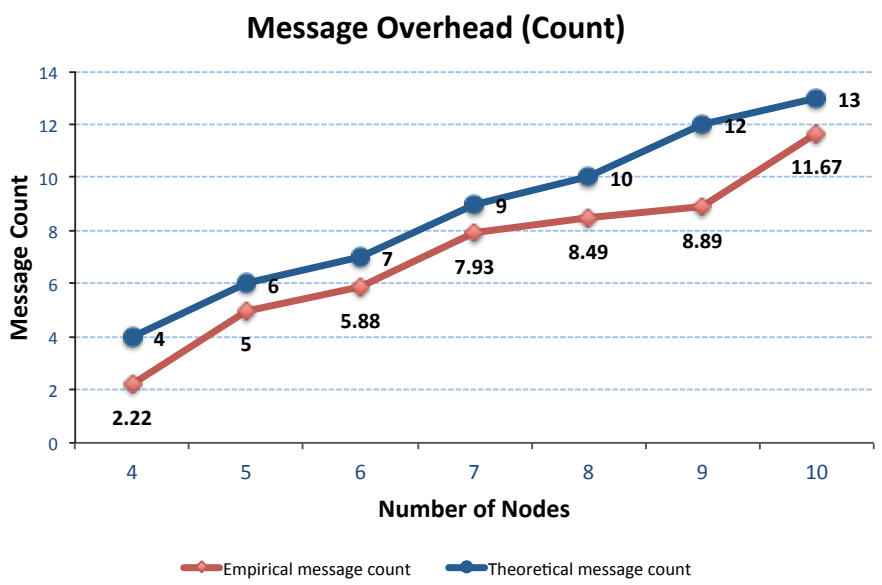

Figure 5.14: Topology Aggregation Message Count

Figure 5.14 illustrates the average amount of *MSG_AR* to be sent for convergence. It is used

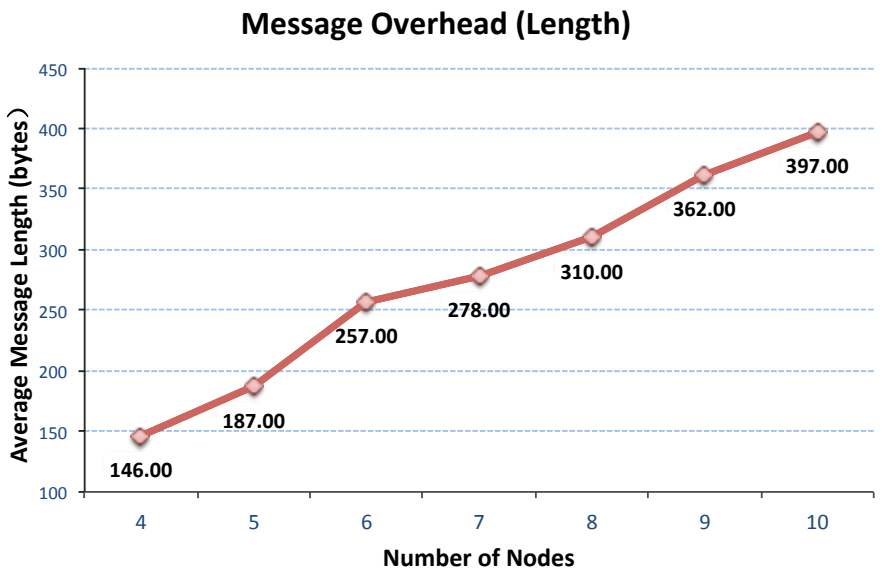

Figure 5.15: Average Message Length

for testing the performance of the flooding mechanism. The AR message counter is defined in the messaging interface. When a new AR message sends to a neighboring domain, it is incremented by 1. Due to converge, we add the AR message counter of each node and get the average value. As we discussed in Chapter 3, the total message overhead for flooding in a graph $G(V, E)$ is $|V| * |E|$. Thus for a single node, the average message count is $|E|$ It is illustrated by the upper blue line, referred to as theoretical message count. The lower red line is the empirical message count. We can reach a conclusion that the flooding performance in our module is better than the standard result.

Figure 5.15 illustrates the average message length in bytes. The message contains *MSG_AR* and *MSG_PV*. It is obtained by summing up all the messages sent prior to converge, then dividing the sum by the total message amount.

5.8 Disjoint Paths Establishment

The experiments of the disjoint paths establishment protocol were conducted in the following setups:

- QoS subscription requests between randomly selected source and destination. This experiment represents a average case.
- QoS subscription requests between given source and randomly selected destinations which simulates multiple subscribers interested in the same publication.

As we mentioned earlier in this Chapter, the set of border nodes are all also considered as edge forwarding engines, so we have the random subscription process generated as follows.

```
static void gen_random(int pubd, int pubn, nr_border)
{
  srand(time(NULL));

  if (!pubd) {
     a[0] = rand() % (total_nodes-1) + 1;
     a[1] = rand() % nr_border;
  } else {
     a[0] = pubd;
     a[1] = pubn;
  }

  a[2] = rand() % (total_nodes-1) + 1;
  a[3] = rand() % nr_border;
}
```

5.8.1 Disjoint Paths Cost

We set up a simple topology to demonstrate that the computed total delay is the minimal total delay. The topology is illustrated as Figure 5.16. We have a three domain topology, for each edge forwarding engine in the subscription domain D^{sub}, a subscription request is issued for any edge forwarding engine in the publication domain D^{pub}. Therefore we have total $|B^{sub}| * |B^{pub}|$ subscription requests. The primary and backup paths costs were recorded. We also implement a centralized algorithm which runs over a full topology with the awareness of all the internal graph in each domain. Finally we compared the path costs results gathered from centralized and distributed algorithm.

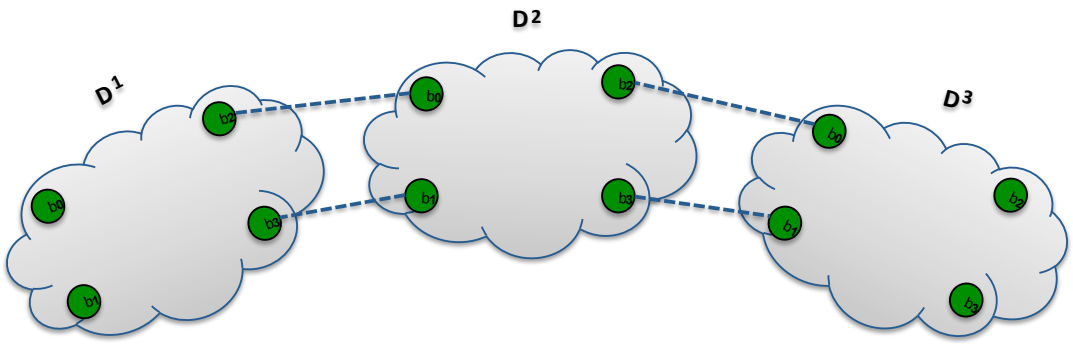

Figure 5.16: Disjoint Paths Cost Setup

Table 5.1: Central-Cost VS Distributed Cost

$DistributedPathCost$	$CentralizedPathCost$
(1,0)-(1,2)-(2,0)-(2,2)-(3,0)-[4]	20-17-10-7-0-[4]
(1,0)-(1,3)-(2,1)-(2,3)-(3,1)-(3,0)-[14]	20-21-22-23-28-26-18-16-8-3-2-1-0-[14]
(1,0)-(1,2)-(2,0)-(2,3)-(3,1)-[8]	26-18-13-12-11-10-7-0-[8]
(1,0)-(1,3)-(2,1)-(2,2)-(3,0)-(3,1)-[10]	26-28-23-22-21-20-17-10-11-12-13-18-16-8-3-2-1-0-[10]
(1,0)-(1,2)-(2,0)-(2,2)-(3,0)-(3,2)-[5]	27-20-17-10-7-0-[5]
(1,0)-(1,3)-(2,1)-(2,3)-(3,1)-(3,2)-[14]	27-23-28-26-18-16-8-3-2-1-0-[14]
(1,0)-(1,2)-(2,0)-(2,2)-(3,0)-(3,3)-[9]	28-23-22-21-20-17-10-7-0-[9]
(1,0)-(1,3)-(2,1)-(2,3)-(3,1)-(3,3)-[9]	28-26-18-16-8-3-2-1-0-[9]
(1,1)-(1,3)-(2,1)-(2,2)-(3,0)-[9]	20-17-13-18-16-8-6-[9]
(1,1)-(1,2)-(2,0)-(2,3)-(3,1)-(3,0)-[10]	20-21-22-23-28-26-18-13-17-10-7-3-4-5-6-[10]
(1,1)-(1,3)-(2,1)-(2,3)-(3,1)-[4]	26-18-16-8-6-[4]
(1,1)-(1,2)-(2,0)-(2,2)-(3,0)-(3,1)-[15]	26-28-23-22-21-20-17-10-7-3-4-5-6-[15]
(1,1)-(1,3)-(2,1)-(2,3)-(3,1)-(3,2)-[10]	27-23-28-26-18-16-8-6-[10]
(1,1)-(1,2)-(2,0)-(2,2)-(3,0)-(3,2)-[10]	27-20-17-10-7-3-4-5-6-[10]
(1,1)-(1,3)-(2,1)-(2,3)-(3,1)-(3,3)-[5]	28-26-18-16-8-6-[5]
(1,1)-(1,2)-(2,0)-(2,2)-(3,0)-(3,3)-[14]	28-23-22-21-20-17-10-7-3-4-5-6-[14]
(1,2)-(2,0)-(2,2)-(3,0)-[3]	20-17-10-7-[3]
(1,2)-(1,3)-(2,1)-(2,3)-(3,1)-(3,0)-[14]	20-21-22-23-28-26-18-16-8-3-7-[14]
(1,2)-(2,0)-(2,3)-(3,1)-[7]	26-18-13-12-11-10-7-[7]
(1,2)-(1,3)-(2,1)-(2,2)-(3,0)-(3,1)-[10]	26-28-23-22-21-20-17-10-11-12-13-18-16-8-3-7-[10]
(1,2)-(2,0)-(2,2)-(3,0)-(3,2)-[4]	27-20-17-10-7-[4]
(1,2)-(1,3)-(2,1)-(2,3)-(3,1)-(3,2)-[14]	27-23-28-26-18-16-8-3-7-[14]
(1,2)-(2,0)-(2,2)-(3,0)-(3,3)-[8]	28-23-22-21-20-17-10-7-[8]
(1,2)-(1,3)-(2,1)-(2,3)-(3,1)-(3,3)-[9]	28-26-18-16-8-3-7-[9]
(1,3)-(2,1)-(2,2)-(3,0)-[8]	20-17-13-18-16-8-[8]
(1,3)-(1,2)-(2,0)-(2,3)-(3,1)-(3,0)-[9]	20-21-22-23-28-26-18-13-17-10-7-3-8-[9]
(1,3)-(2,1)-(2,3)-(3,1)-[3]	26-18-16-8-[3]
(1,3)-(1,2)-(2,0)-(2,2)-(3,0)-(3,1)-[14]	26-28-23-22-21-20-17-10-7-3-8-[14]
(1,3)-(2,1)-(2,3)-(3,1)-(3,2)-[9]	27-23-28-26-18-16-8-[9]
(1,3)-(1,2)-(2,0)-(2,2)-(3,0)-(3,2)-[9]	27-20-17-10-7-3-8-[9]
(1,3)-(2,1)-(2,3)-(3,1)-(3,3)-[4]	28-26-18-16-8-[4]
(1,3)-(1,2)-(2,0)-(2,2)-(3,0)-(3,3)-[13]	28-23-22-21-20-17-10-7-3-8-[13]

From Table 5.1, we can see that the disjoint paths cost computed in inter-domain routing module are identical to the paths cost computed by the centralized disjoint paths algorithm.

5.8.2 Paths Establishment Delay

The Disjoint Establishment Delay is defined as below. We choose random subscriber to establishment the paths with delay in milliseconds. Figure 5.17 illustrates the path establishment delay.

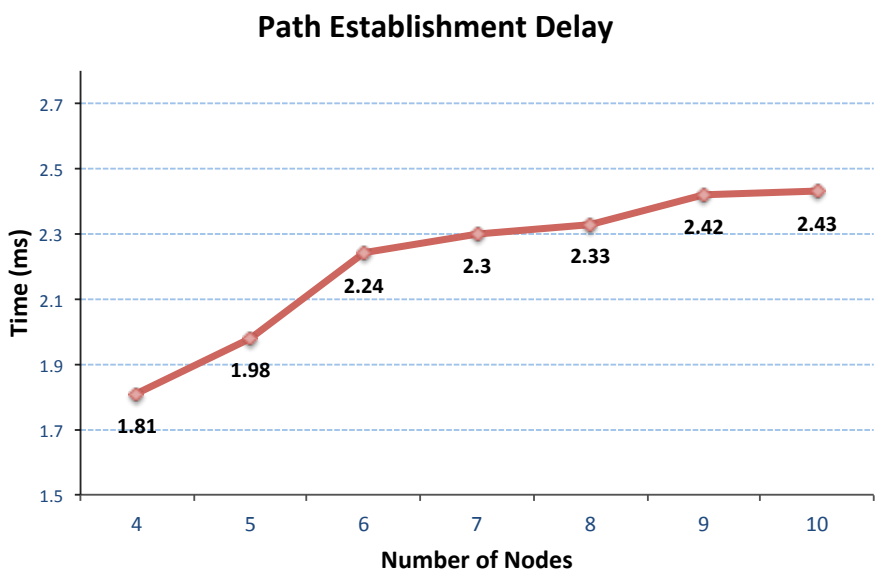

Figure 5.17: Path Establishment Delay

5.8.3 Fault Tolerant Experiment

This is an essential feature of the inter-domain routing model. To evaluate the feature, we design the experiment which demonstrates the system resilience of the inter-domain routing module.

In the current management plane framework if a broker is crashes, all its children brokers fail to establish inter-domain disjoint paths. In our P2P framework, if a broker is crashes, the adjacent broker will learn the situation and broadcast the bad news over all the domains and thus the

failed one is removed from the routing tables of all these domains. Consequently the subscription requests involving the filed domain are denied. Moreover, the path establishment process is not affected as long as there are disjoint paths between a subscriber and a publisher. The end-to-end delay bound is presumably to be larger if the minimal total cost disjoint paths traverse the previous failed domain. Anyhow the minimal total paths cost is computed based on all the active (not failed) domains.

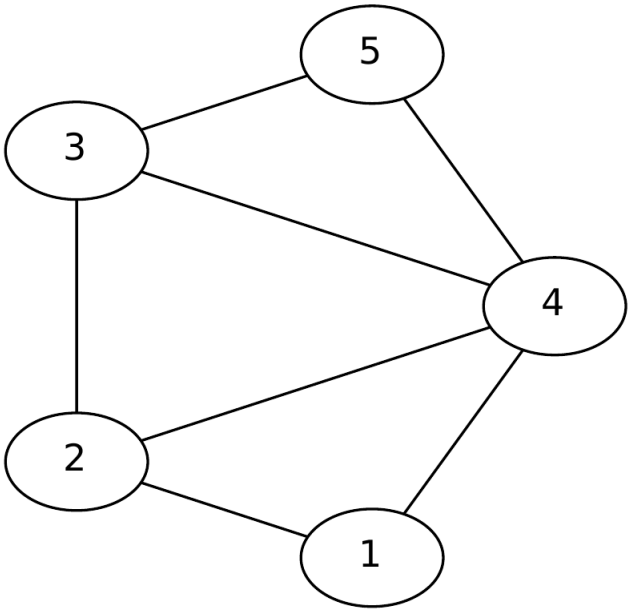

Figure 5.18: Fault Tolerant 5 Node Topology

The implementation of automatically detecting failures and restoring to the consistent network routing state is in progress. We designed an experiment to simulate the failed domain and it was conducted as follows. We begin from a five-node topology as shown in Figure 5.18. Then the disjoint paths (1,0)-(5,3) was computed. To simulate the failure of a domain, each time we manually remove these domains 2, 3, 4 respectively. Then based on the topology in which domains are converged again, the same end-to-end (1,0)-(5,3) disjoint paths were computed.

The original computed disjoint paths are:

(1,0)−(4,2)−(4,0)−(5,2)−(5,3)−[8]
(1,0)−(1,3)−(2,1)−(3,3)−(5,1)−(5,3)−[9]

After removing the domain 2, the disjoint paths are

(1,0)−(1,2)−(4,0)−(4,2)−(5,0)−(5,3)−[9]
(1,0)−(1,3)−(4,1)−(3,3)−(5,3)−[9]

After removing the Domain 3, the computed paths are

(1,0)−(4,2)−(4,0)−(5,2)−(5,3)−[8]
(1,0)−(1,2)−(2,0)−(4,2)−(4,1)−(5,3)−[9]

After removing the domain 4, the computed paths are:

(1,0)−(1,3)−(2,1)−(3,3)−(3,1)−(5,3)−[9]
(1,0)−(1,2)−(2,0)−(3,2)−(3,0)−(5,2)−(5,3)−[10]

As an result our inter-domain routing module enhances the system resilience in GridStat.

CHAPTER SIX

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The current GridStat routing architecture is based on a hierarchical tree structure in the *management plane*. The centralized architecture performs well in the case where subscriptions are mainly confined to lower levels of the management hierarchy. However, as the inter-domain path computations must be made at the level where the non-leaf QoS broker has the full end-to-end visibility, the architecture obviously does not scale well in a large network. It also suffers from single node failure. If any non-leaf broker crashes, it will affects the inter-domain path finding of all its children domains.

Based on these observations, we proposed a peer-to-peer architecture to take advantage of distributed benefits. We presented a peer-to-peer routing module using two protocols: *path vector protocol* and *path establishment protocol*. Our approach enhances the GridStat *management plane* in terms of flexibility and resilience. For flexibility, it allows brokers to discover each other automatically and then exchange necessary information to be able to compute disjoint paths. For resilience, brokers are able to learn new end-to-end disjoint paths after a broker fails. We also use an *aggregation topology* technique to not only conceal the detailed intra-domain topology, but also reduce the message overhead with the assumption that border forwarding engines are only a small subset of total forwarding engines in the data plane. The path establishment protocol distributes inter-domain disjoint paths computation across the set of domains consisting of the *primary path*. We have also implemented a inter-domain routing module prototype. Based on the simulation results, we show that the performance of the inter-domain routing architecture is scalable and the single node failure is resolved so it improves the fault tolerance in GridStat management plane.

6.2 Future Work

The inter-domain disjoint paths computation is based on the algorithm proposed in [29]. The problem is known as *Min-Sum* and guarantees the minimal total weight of primary and backup paths. A relevant problem known as *Min-Max* is described as follows: Finding two disjoint paths that minimize $max(W(P_1), W(P_2))$. It clearly improves the computed disjoint paths quality. With *Min-Max* the path evaluation phase in path establishment protocol can be completely removed, since we would then be able to evaluate the paths weights in the publication domain. However this problem is NP-hard [32]. The heuristic for *Min-Max* remains for the future work. Furthermore, the Min-Sum problem focuses on the link disjoint paths. To obtain more fault tolerant feature on node level, incorporating node disjoint paths mechanism into the proposed framework needs further investigation.

In addition to *Min-Max* problem, the proposed disjoint paths computation performs sequentially. Each subscription request starts from subscription domain and ends in the publication domain, followed by the path reservation message from publication domain to subscription. However due to the multicast characteristic in GridStat, the multicast feature should take into account in the future work, especially with a set of heuristics proposed by Irava's [22]. It allows new destination joins the multicast tree with satisfying the delay requirements while keep the multicast tree cost low, in addition, it enables end-to-end reliable routes (disjoint paths). However the heuristics are based on the full topology of the data plane network and performs in a centralized fashion. More research needs to conduct on developing a distributed version.

This thesis focuses on the inter-domain routing architecture and corresponding protocols. The security issues of the P2P management plane architecture must be addressed, such as entity authentication, data encryption/decryption and integrity.

APPENDIX ONE
ALGORITHMS

A.1 Centralized Link-Disjoint Path algorithm

The concise steps of the Link-Disjoint Paths Algorithm are listed as follows. Input is a graph $G(V, E)$ with vertices set V and edge set E.

1. Find the shortest path P_1 from source s to destination t by running Dijkstra's algorithm
2. For each link $e \in P_1$, reverse it and negate the weight
3. Find the shortest path P_2 in the remaining Graph $\tilde{G}$
4. Discard links that appear in P_1 and P_2 in opposite direction
5. From the remaining links P_1 and P_2, return two disjoint (s, t) paths $\hat{P}_1$ and $\hat{P}_2$

A.2 Distributed Link-Disjoint Path algorithm

Algorithm FINDAR *(D_i, B_i):*

Input:

D_i - a routing domain,
B_i - the set of border nodes of D_i.

Output:

The aggregated representation $A_i = \{M_i'\} \cup \{M_i^{j,l} \mid b_j \in B_i, b_l \in B_i\}$ of D_i

for each two border nodes b_j and b_l of D_i **do**
    Compute a shortest path $P_i^{j,l}$ between b_j and b_l in D_i;
    $M_i'(j, l) \leftarrow W(P_i^{j,l})$
    Construct an auxiliary graph $D_i^{j,l}$ formed from D_i by reversing all links of $P^{j,l}$ and negating their weights
    for each two border nodes b_x and b_y of D_i **do**
        Compute a shortest path $P_i^{j,l}(x, y)$ between b_x and b_y in $D_i^{j,l}$
        $M_i^{j,l}(x, y) \leftarrow W(P_i^{j,l}(x, y))$.

Algorithm FIND2DP *(E^{inter}, $\{A_i\}$):*

Input:

E^{inter} - a set of the inter-domain links,

For each routing domain D_i

$A_i = \{M'_i\} \cup \{M_i^{j,l} \mid b_j \in B_i, b_l \in B_i\}$ -

The aggregated representation of D_i.

Output:

An auxiliary network $G'(V', E')$ and two paths P_1 and P_2 in G'.

$V' \leftarrow V(D^s) \cup \{B_i \mid D_i \in G\}$

$E' \leftarrow E(D^s) \cup E^{inter}$

for each routing domain D_i of G **do**

  for each two border nodes b_j and b_l of D_i **do**

    $E' \leftarrow E' \cup (b_j, b_l)$;

    $w_{(b_j, b_l)} \leftarrow M'_i(j, l)$

Find a shortest path P_1 between s and t in $G'(V', E')$

Reverse all inter-domain links and links that belong to D^s in P_1 and negate their weight

For each routing domain D_i of G except D^s**do**

  if P_1 contains a link (b_j, b_l) that connects border nodes of D_i **then**

    for each two border nodes b_x and b_y of D_i **do**

      $w_{(b_x, b_y)} \leftarrow M_i^{j,l}(x, y)$

Find a shortest path P_2 between s and t in $G'(V', E')$.

APPENDIX TWO
MESSAGE FORMAT

- Common Header

```
<Common Header> ::= <source_id>
                    <msg_contained>
                    <msg_length>
```

- MSG_PV

```
<MSG_PV> ::= <Common Header>
             <msg_type>
             <num_of_updates>
             <PV-list>


<PV-list> ::= <Path Vector>
              [ PV-list ]

<Path Vector> ::= <seq_num>
                  <age>
                  <dest>
                  <cost>
                  <num_of_nodes>
                  <path>
```

- MSG_AR

```
<MSG_AR> ::= <Common Header>
             <msg_type>
             <num_of_updates>
             <AR-list>

<AR-list> ::= <Topology Aggregation>
              [ AR-list ]


<Topology Aggregation> ::= <ar_length>
                           <seq_num>
```

```
                                <age>
                                <ar_domain_id>
                                <ar_vertex_num>
                                <ar_edge_num>
                                <ar_edges>
```

- MSG_AR_AUX

```
<MSG_AR_AUX> ::= <Common Header>
             <msg_type>
             <num_of_updates>
             <AR_AUX_list>

<AR_AUX-list> ::= <Topology Aggregation>
             [ AR_AUX-list ]


<Topology Aggregation> ::= <ar_aux_length>
                           <seq_num>
                           <age>
                           <ar_aux_domain_id>
                           <ar_aux_vertex_num>
                           <ar_aux_edge_num>
                           <ar_aux_edges>
```

- MSG_PATH

```
<MSG_PATH> ::= <Common Header>
               <next_hop_node>
               <req_time_sec>
               <req_time_usec>
               <explict_path>
               [<MSG_AR_AUX>]

<explicit_path> ::= <explicit_path_length>
                    <msg_type>
                    <number_of_nodes>
                    <nodelist>
```

- MSG_RSVP

```
<MSG_RSVP> ::= <Common Header>
               <next_hop_node>
               <req_time_sec>
               <req_time_usec>
               <explict_path>

<explicit_path> ::= <explicit_path_length>
                    <msg_type>
                    <number_of_nodes>
                    <nodelist>
```

- MSG_PV_ACK

```
<MSG_PV_ACK> ::= <Common Header>
                 <msg_type>
                 <ack_list>
```

- MSG_AR_ACK

```
<MSG_AR_ACK> ::= <Common Header>
                 <msg_type>
                 <ack_list>
```

APPENDIX THREE
INTERFACE

- data-plane.h

```
typedef struct graph *graph_t;
typedef struct path *path_t;
typedef struct global_t *global_t;

graph_t  graph_init(int id, int V, int E, int B);

graph_t  graph_scan(char *msg, graph_t G, int aux);

void   graph_show(graph_t G);

void   graph_showsp(path_t p);

path_t graph_sp(graph_t G, int s, int d);

graph_t  graph_clone(graph_t G);

void   graph_insert_edge(graph_t G, Edge *e);

void graph_free(graph_t G);

graph_t graph_compute_ar(graph_t G);

graph_t graph_reverse_negate(graph_t G, path_t path);

void graph_show_ar(graph_t G);

void graph_path_free(path_t p);

int *graph_get_top_agg(graph_t G);

short int *graph_get_interlink(graph_t G);

graph_t graph_build_global (graph_t G, int *dpar,
```

```
                                int len, int aux);

graph_t graph_construct_global_graph();

int *graph_path_get_ginfo(path_t p, int *len);

int *graph_get_aux_ar(graph_t G, int x, int y,
                      int *len);

int *graph_get_global_path (graph_t G, path_t p,
                            int *len, int reverse);
```

- management-plane.h

```
int mp_send_ar_ack(int ackto_id);

int mp_init_neighbor_connect(int id);

int mp_flooding();

graph_t mp_construct_global_graph();

void mp_init(int id, int port, int total, int nneighbor);

void mp_scan(void);

void mp_open_server();

int mp_get_it_msg(int sendtoid, uint8_t *buf, int *len);

int mp_to_update_ar (int domainid, int seq_num,
                     int recvid);

int mp_store_msg_it(void *msg_inter, int msginterlen,
                    short *interarr);

int mp_store_msg_dp(uint8_t *buf, int msgdplen, int *dp,
                    int type, int seq_num, int age);

void mp_update_ack (int ackid, int recvid,
```

```
                    uint16_t msg_type);
```

- msg.h

```
#define MSG_PV         0x1
#define MSG_AR         (MSG_PV << 1)
#define MSG_PUB        (MSG_PV << 2)
#define MSG_PV_ACK     (MSG_PV << 3)
#define MSG_AR_ACK     (MSG_PV << 4)
#define MSG_PATH       (MSG_PV << 5)
#define MSG_RESV       (MSG_PV << 6)
#define MSG_EVAL       (MSG_PV << 7)
#define MSG_EVAL_ACK   (MSG_PV << 8)
#define MSG_PUB_TEAR   (MSG_PV << 9)
#define MSG_SUB_TEAR   (MSG_PV << 10)

#define PV_NUM_OFFSET sizeof(short)
#define PATH_VECTOR_OFFSET 4

/* offset for ar header */
#define ARS_NUM_OFFSET sizeof(short)
#define ARS_LEN_OFFSET sizeof(int)
#define ARS_MSG_OFFSET (sizeof(int) * 2)

/* offset for a single ar msg */
#define AR_LEN_OFFSET 0x0
#define AR_SEQ_OFFSET  (AR_LEN_OFFSET + sizeof(int))
#define AR_AGE_OFFSET  (AR_SEQ_OFFSET            \
                        + sizeof(short))
#define AR_DOMAINID_OFFSET (AR_AGE_OFFSET        \
                            + sizeof(short))
#define AR_VERTEX_NUM_OFFSET (AR_DOMAINID_OFFSET \
                            + sizeof(int))
#define AR_EDGE_NUM_OFFSET (AR_VERTEX_NUM_OFFSET \
                            + sizeof(int))
#define AR_EDGES_OFFSET (AR_EDGE_NUM_OFFSET      \
                            + sizeof(int))

int msg_send (int nodeid, int sendtoid,
              unsigned short msg_type, gps_t path_seg,
```

```
              uint8_t *msg, int msglen);

uint8_t *msg_construct_pv(uint8_t *buf, int sendtoid);

uint8_t *msg_construct_ack(uint8_t *buf, int sendtoid,
 uint16_t meg_tosend,
                                uint16_t msg_tochk);

uint8_t *msg_construct_ar (uint8_t *buf,
                           int sendtoid);

uint8_t *msg_construct_inter (uint8_t *buf,
                              int sendtoid);

uint8_t *msg_construct_path (uint8_t *buf,
                             gps_t path_seg,
                             short msg_type);

uint8_t *msg_construct_auxar (uint8_t *buf,
                              gps_t path_seg,
                              uint8_t *pathstart);

uint8_t *msg_process_pv_ack (uint8_t *buf, int recvid);

uint8_t *msg_process_pv (uint8_t *buf, int recvid);

uint8_t *msg_process_ar (uint8_t *buf, int recvid);

uint8_t *msg_process_inter(uint8_t *buf);

uint8_t *msg_process_path(uint8_t *buf, int recvid);

uint8_t *msg_process_ack(uint8_t *buf, int recvid);

static long pv_seq = 1;
static long ar_seq = 1;
```

- path-vector.h

```
int pv_update_path_vector(int neighbor_id,
```

```
                          path_vector_t *pv);

void pv_update_ack (int ackid, int recvid);

int pv_process_pv (int neighbor_id,
                   path_vector_t *pv);

int pv_send_ack(int ackto_id);

int pv_flooding();
```

- path-disjoint.h

```
int *pd_get_explicit_path (path_t prim_path, int *next_hop,
                           int *len, int *innode,
                           int *enode, graph_t G,
                           short msg_type);

uint8_t *pd_get_gps_msg (uint8_t *buf, gps_t t,
                                 short msg_type);

uint8_t *pd_get_aux_ar_msg (uint8_t *buf, gps_t t);

int pd_find_pathseg_info(int *path, int len,
                         int *innode, int *enode);

int pd_compute_pri_path (int srcdomain, int srcnode,
                         int destdomain, int destnode,
                         graph_t Gglobal, path_t *path);

gps_t pd_init_gps (path_t p, int *path, int nnodes,
                   uint8_t *buf, int buflen, int exp_len,
                   short msg_type, graph_t G)

graph_t pd_construct_aux_graph(gps_t t, path_t *path);

void pd_construct_msg_rsvp (graph_t G, path_t p, gps_t old);

int pd_disjoint_handle_req (int *path, int nnodes,
                            int recvid, uint8_t *buf,
```

```
                                int buflen, int exp_len,
                                short msg_type);

uint8_t *pd_construct_explicit_path(uint8_t *buf,
                                    int *path, int nnodes,
                                    short msg_type);

int pd_path_esb(int srcd, int srcn, int dstd, int dstn);
```

BIBLIOGRAPHY

[1] D. E. Bakken, C. H. Hauser, H. Gjermundrod, and A. Bose. Towards more flexible and robust data delivery for monitoring and control of the electric power grid. `http://gridstat.net/trac/raw-attachment/wiki/WikiStart/GridStat-EECS-GS-003.pdf/`.

[2] Deepankar Medhi Karthikeyan Ramasamy. *Network Routing Algorithms, Protocols, and Architectures*. Morgan Kaufmann, 2007.

[3] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031 (Proposed Standard), January 2001. Updated by RFCs 6178, 6790.

[4] D. Smith, J. Mullooly, W. Jaeger, and T. Scholl. Label Edge Router Forwarding of IPv4 Option Packets. RFC 6178 (Proposed Standard), March 2011.

[5] K. Kompella, J. Drake, S. Amante, W. Henderickx, and L. Yong. The Use of Entropy Labels in MPLS Forwarding. RFC 6790 (Proposed Standard), November 2012.

[6] E. Mannie. Generalized Multi-Protocol Label Switching (GMPLS) Architecture. RFC 3945 (Proposed Standard), October 2004. Updated by RFC 6002.

[7] L. Berger and D. Fedyk. Generalized MPLS (GMPLS) Data Channel Switching Capable (DCSC) and Channel Set Label Extensions. RFC 6002 (Proposed Standard), October 2010.

[8] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209 (Proposed Standard), December 2001. Updated by RFCs 3936, 4420, 4874, 5151, 5420, 5711, 6780, 6790.

[9] K. Kompella and J. Lang. Procedures for Modifying the Resource reSerVation Protocol (RSVP). RFC 3936 (Best Current Practice), October 2004.

[10] A. Farrel, D. Papadimitriou, J.-P. Vasseur, and A. Ayyangar. Encoding of Attributes for Multiprotocol Label Switching (MPLS) Label Switched Path (LSP) Establishment Using Resource ReserVation Protocol-Traffic Engineering (RSVP-TE). RFC 4420 (Proposed Standard), February 2006. Obsoleted by RFC 5420.

[11] CY. Lee, A. Farrel, and S. De Cnodder. Exclude Routes - Extension to Resource ReserVation Protocol-Traffic Engineering (RSVP-TE). RFC 4874 (Proposed Standard), April 2007. Updated by RFC 6001.

[12] A. Farrel, A. Ayyangar, and JP. Vasseur. Inter-Domain MPLS and GMPLS Traffic Engineering – Resource Reservation Protocol-Traffic Engineering (RSVP-TE) Extensions. RFC 5151 (Proposed Standard), February 2008.

[13] A. Farrel, D. Papadimitriou, JP. Vasseur, and A. Ayyangarps. Encoding of Attributes for MPLS LSP Establishment Using Resource Reservation Protocol Traffic Engineering (RSVP-TE). RFC 5420 (Proposed Standard), February 2009. Updated by RFC 6510.

[14] JP. Vasseur, G. Swallow, and I. Minei. Node Behavior upon Originating and Receiving Resource Reservation Protocol (RSVP) Path Error Messages. RFC 5711 (Proposed Standard), January 2010.

[15] L. Berger, F. Le Faucheur, and A. Narayanan. RSVP ASSOCIATION Object Extensions. RFC 6780 (Proposed Standard), October 2012.

[16] A. Farrel, J.-P. Vasseur, and J. Ash. A Path Computation Element (PCE)-Based Architecture. RFC 4655 (Informational), August 2006.

[17] JP. Vasseur and JL. Le Roux. Path Computation Element (PCE) Communication Protocol (PCEP). RFC 5440 (Proposed Standard), March 2009.

[18] Kjell Harald Gjermundrod. *Flexible QoS-Managed Status Dissemination Middleware Framework For The Electric Power Grid*. PhD thesis, Washington State University, Aug 2006.

[19] Jeffrey M Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14(1):95–116, 1984.

[20] Zheng Wang and Jon Crowcroft. *QoS routing for supporting resource reservation*, volume 14. 1996.

[21] Shigang Chen and Klara Nahrstedt. On finding multi-constrained paths. In *Communications, 1998. ICC 98. Conference Record. 1998 IEEE International Conference on*, volume 2, pages 874–879. IEEE, 1998.

[22] Venkata Srinivas Irava. *Low-Cost Delay-Constrained Multicast Routing Heuristics And Their Evaluation*. PhD thesis, Washington State University, Aug 2006.

[23] F. Ricciato, U. Monaco, and D. Ali. *Distributed schemes for diverse path computation in multidomain MPLS networks*, volume 43. 2005.

[24] J.P. Lang, Y. Rekhter, and D. Papadimitriou. RSVP-TE Extensions in Support of End-to-End Generalized Multi-Protocol Label Switching (GMPLS) Recovery. RFC 4872 (Proposed Standard), May 2007. Updated by RFC 4873.

[25] J. W. Suurballe and R. E. Tarjan. A quick method for finding shortest pairs of disjoint paths networks. *Networks*, 1(2):325–336, 1984.

[26] N. Bitar JP. Vasseur, R. Zhang and JL. Le Roux. A Backward-Recursive PCE-Based Computation (BRPC) Procedure to Compute Shortest Constrained Inter-Domain Traffic Engineering Label Switched Paths. RFC 5441 (Proposed Standard), April 2009.

[27] I. Nishioka and D. King. Use of the Synchronization VECtor (SVEC) List for Synchronized Dependent Path Computations. RFC 6007 (Informational), September 2010.

[28] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFCs 6286, 6608, 6793.

[29] A. Sprintson, M. Yannuzzi, A. Orda, and X. Masip-Bruin. *Reliable Routing with QoS Guarantees for Multi-Domain IP/MPLS Networks*. 2007.

[30] University of Utah. Emulab. `http://www.emulab.net/`.

[31] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. October 2010.

[32] Chung-Lun Li, S.Thomas McCormick, and David Simchi-Levi. The complexity of finding two disjoint paths with min-max objective function. *Discrete Applied Mathematics*, 26(1):105 – 115, 1990.