

**A RESILIENT REDUNDANT ROUTING FRAMEWORK  
FOR FUTURE INTERNET ARCHITECTURE**

by

Hristo Asenov

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Spring 2016

© 2016 Hristo Asenov  
All Rights Reserved

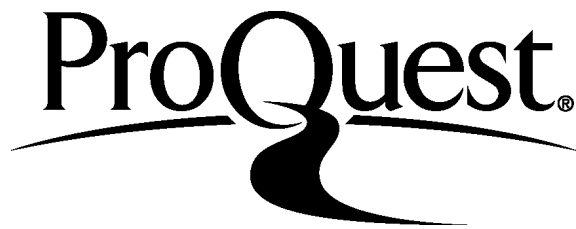
ProQuest Number: 10156576

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10156576

Published by ProQuest LLC (2016). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code  
Microform Edition © ProQuest LLC.

ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

**A RESILIENT REDUNDANT ROUTING FRAMEWORK  
FOR FUTURE INTERNET ARCHITECTURE**

by

Hristo Asenov

Approved: \_\_\_\_\_  
Kathleen F. McCoy, Ph.D.  
Chair of the Department of Computer and Information Sciences

Approved: \_\_\_\_\_  
Babatunde A. Ogunnaike, Ph.D.  
Dean of the College of Engineering

Approved: \_\_\_\_\_  
Ann L. Ardis, Ph.D.  
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_

Chase Cotton, Ph.D.  
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_

Chien-Chung Shen, Ph.D.  
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_

Vasil Hnatyshin, Ph.D.  
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_

John Cavazos, Ph.D.  
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_

Robert Broberg, BS ChE.

Member of dissertation committee

## ACKNOWLEDGEMENTS

To Dr. Chase Cotton, thank you for helping and guiding me through my academic and research progress. Only with your advice and support was this work made possible. I could always rely on you, and you always believed in me and my work. To Dave Sincoskie, thank you for seeing potential and motivation in me as an incoming graduate student.

To Leslie, thank you for your support and encouragement. As my fiance, you are the best life partner I could ask for. I know that the water currents have not always been calm, but we made it got through every challenge we faced. I would like to thank my father, Stilian Asenov, for always being by my side, and for raising me into the person that I am today.

To Dr. Fouad Kiamilev, thank you for always encouraging me and being such a positive presence throughout all of my PhD. You provided me with research space and allowed me to explore my interests in Electrical Engineering. I appreciate all the helpful advise you provided when I needed it.

I would like to thank Dr. Vasil Hnatyshin for all his encouragement and support when applying to graduate school, and later on during my studies. He is a great individual and I am very lucky to have worked with him.

To Chien-Chung Shen, I appreciate the help, advice and support for this work. As my co-advisor, your suggestions into conferences and research topics were invaluable.

Thanks for all the help and support from everyone at CVORG. Thanks to Stephen Sigwart for being a good friend and collaborator. Thanks to Fatema for being so friendly and always willing to discuss her work.

This work was supported by Project NEBULA under NSF CNF funding #1040614 and the LUDD project at Cisco Systems, Inc.

## TABLE OF CONTENTS

<b>LIST OF TABLES</b> . . . . .	<b>xi</b>
<b>LIST OF FIGURES</b> . . . . .	<b>xii</b>
<b>ABSTRACT</b> . . . . .	<b>xiv</b>
 <b>Chapter</b>	
<b>1 INTRODUCTION</b> . . . . .	<b>1</b>
1.1 Motivation . . . . .	2
1.2 First Generation: Single Bus Architecture with Centralized Route Processing . . . . .	2
1.3 Second Generation: Distributed Route Caching . . . . .	3
1.4 Third Generation: Switched Backplane with Forwarding Engine . . . . .	4
1.5 Next Generation: Distributed Routers . . . . .	4
1.6 Contributions . . . . .	5
1.7 Dissertation Organization . . . . .	5
<b>2 BACKGROUND AND KEY CONCEPTS</b> . . . . .	<b>6</b>
2.1 Process Groups . . . . .	6
2.2 Scaling . . . . .	6
2.3 Different High Availability application models . . . . .	7
2.3.1 Load-Sharing/Stateless . . . . .	7
2.3.2 Active-Standby . . . . .	8
2.3.3 N-Modular Redundancy . . . . .	9
2.4 Failure Scenarios . . . . .	10
2.5 Service Models . . . . .	10
2.5.1 Let It Crash Model . . . . .	11
2.6 High Availability . . . . .	11

2.7	Consistency . . . . .	11
2.7.1	Sequential Consistency . . . . .	12
2.7.2	Linearizability . . . . .	13
2.8	CAP Theorem . . . . .	15
2.9	Strongly Consistent Systems . . . . .	17
2.10	Weakly/Eventually Consistent System . . . . .	18
<b>3</b>	<b>PRIOR WORK . . . . .</b>	<b>19</b>
3.1	Safety-Critical Systems . . . . .	20
3.1.1	Automotive Applications . . . . .	20
3.1.2	Aerospace and Spacecraft . . . . .	21
3.2	Consensus Protocols . . . . .	23
3.3	Weaker Forms of Consistency . . . . .	26
3.3.1	Virtual Synchrony . . . . .	26
3.3.2	Distributed Hash Table . . . . .	27
3.4	Software Defined Networks . . . . .	31
3.5	Next Generation Routing Architectures . . . . .	33
<b>4</b>	<b>BUILDING BLOCKS . . . . .</b>	<b>41</b>
4.1	Requirements . . . . .	41
4.2	Splitting and Voting with N-Modular Redundancy . . . . .	42
4.2.1	Voting Implementations . . . . .	43
4.2.2	Different Voting Strategies . . . . .	45
4.2.3	No Agreement . . . . .	46
4.3	Routing Information Base . . . . .	47
4.3.1	Routing Table Structure . . . . .	47
4.3.2	N-Modular Redundant Model . . . . .	48
4.3.3	RIB Update Monitoring . . . . .	49

4.3.4	RIB Sibling Synchronization . . . . .	50
4.4	HA Mechanisms . . . . .	51
4.4.1	Checkpoint/Recovery . . . . .	51
4.4.2	Resource Registry . . . . .	53
4.4.3	Synchronization . . . . .	53
4.5	Distributed Key-Value Stores . . . . .	54
<b>5</b>	<b>GENERAL ARCHITECTURAL FRAMEWORK . . . . .</b>	<b>56</b>
5.1	System Model . . . . .	56
5.2	Proposed Platform . . . . .	58
5.2.1	Comparison to SDNs . . . . .	59
5.3	Control Plane . . . . .	60
5.4	Data Plane . . . . .	60
5.4.1	Route Bricks . . . . .	61
5.4.2	NetSlice . . . . .	61
5.5	Separation Between Control Plane and Data Plane . . . . .	62
5.6	SIS-IS . . . . .	62
5.7	Different Fate Sharing Domains . . . . .	65
5.7.1	Process Migration . . . . .	66
5.8	HARRY . . . . .	68
<b>6</b>	<b>COMPONENTS AND CAPABILITIES . . . . .</b>	<b>69</b>
6.1	Components . . . . .	69
6.1.1	A Typical Deployment . . . . .	71
6.2	Capabilities . . . . .	73
6.2.1	Failure Detection . . . . .	74
6.2.2	Leader Election . . . . .	75
6.2.3	Sibling Restarts . . . . .	76

6.2.4	Consistency . . . . .	78
6.2.4.1	Weak Consistency at Siblings . . . . .	78
6.2.4.1.1	Checkpointing . . . . .	78
6.2.4.1.2	State Replay . . . . .	80
6.2.4.1.3	State Catchup . . . . .	80
6.2.4.2	Strong Consistency at Controller - Monotonically Increasing ID . . . . .	81
<b>7</b>	<b>EVALUATION . . . . .</b>	<b>84</b>
7.1	Resource Registry . . . . .	84
7.2	Implementation Overhead . . . . .	88
7.2.1	Overhead in Installing a RIB Entry . . . . .	88
7.2.2	Overhead Over Default Configuration . . . . .	89
7.3	Recovery Time . . . . .	92
7.3.1	Recovery Time of Single Sibling from Single Fault . . . . .	92
7.3.2	Smallest Possible Fault Interarrival Period . . . . .	93
7.4	Measuring Responsiveness (using ALL or ANY voter) . . . . .	94
<b>8</b>	<b>CONCLUSION . . . . .</b>	<b>96</b>
8.1	Contributions . . . . .	96
8.2	Lessons Learned . . . . .	97
8.3	Future Work . . . . .	98
8.3.1	Checkpoint Collapsing . . . . .	98
8.3.2	Transparent HA framework . . . . .	99
8.3.3	Alternative Voting Implementations . . . . .	99
8.4	Final Remarks . . . . .	100
	<b>BIBLIOGRAPHY . . . . .</b>	<b>101</b>

## Appendix

<b>A IMPLEMENTATION DETAILS . . . . .</b>	<b>112</b>
A.1 RouteFlow . . . . .	112
A.2 Controller Initialization with Zebralite . . . . .	114
A.3 Sibling Initialization with Controllers . . . . .	115

## LIST OF TABLES

2.1	SubHistory Of Executions . . . . .	16
2.2	Composition of Histories . . . . .	16
3.1	Table of Prior Work . . . . .	39
7.1	SIS-IS Registration Delays (ms) . . . . .	84
7.2	SIS-IS Deregistration Delays (ms) . . . . .	84
7.3	Riak Registration Delays (ms) . . . . .	85
7.4	Riak Deregistration Delays (ms) . . . . .	85
7.5	Path Switching (in sec) . . . . .	91
7.6	Different Voting Strategies (in msec) . . . . .	95

## LIST OF FIGURES

2.1	No Sequential Consistency . . . . .	12
2.2	Sequential Consistency . . . . .	13
2.3	Sequential Consistency, Writes and Reads . . . . .	14
2.4	Linearization . . . . .	15
2.5	CAP Theorem . . . . .	17
3.1	Ring with Consistency Hashing . . . . .	28
3.2	R and W = 1 . . . . .	30
3.3	R and W = ALL . . . . .	31
4.1	Processing Stages . . . . .	43
4.2	Voter on Three Sibling Streams . . . . .	44
4.3	Voter Strategies: N is 7 and n is 5 . . . . .	46
4.4	Sample Routing Table . . . . .	48
4.5	Process Structure Model . . . . .	49
4.6	Change of Routes . . . . .	50
4.7	DHT and Centralized Server . . . . .	53
5.1	Core Router Line Card Model . . . . .	57
5.2	Routing Architecture Design . . . . .	59
5.3	SISIS Architecture . . . . .	64

5.4	SISIS Address Format . . . . .	65
5.5	Process Migration . . . . .	67
6.1	Different Parts of HARRY . . . . .	70
6.2	HA System in Operation . . . . .	73
6.3	Process Timeout . . . . .	75
6.4	Checkpointing . . . . .	79
6.5	Strongly Consistent Coordination . . . . .	81
7.1	SIS-IS RIB - Average Time To Search . . . . .	88
7.2	DHT - Average Time To Search . . . . .	89
7.3	Percent Increase . . . . .	90
7.4	R1 Router Switching . . . . .	91
7.5	GNS3 Architecture . . . . .	91
7.6	Time to ReSync . . . . .	92
7.7	Minimum Fault Interarrival Time . . . . .	94
A.1	Controller Initialization With Zebralite . . . . .	114
A.2	Controller Initialization With Siblings . . . . .	116
A.3	Controller Client State Diagram . . . . .	117
A.4	Sibling Ctrl State Diagram at Siblings . . . . .	117

## ABSTRACT

The unending expansion of Internet-enabled services drives demand for higher speeds and more stringent reliability capabilities in the commercial Internet. Network carriers, driven by these customer demands, increasingly acquire and deploy ever larger routers with an ever growing feature set. The software however, is not currently structured adequately to preserve Non-Stop Operation and is sometimes deployed in non-redundant software configurations.

In this dissertation, we look at current Routing Frameworks and address the growing reliability concerns of Future Internet Architectures. We propose and implement a new distributed architecture for next generation core routers by implementing N-Modular Redundancy for Control Plane Processes. We use a distributed key-value data store as a substrate for High Availability (HA) operation and synchronization with different consistency requirements. A new platform is proposed, with many characteristics similar to Software Defined Network, on which the framework is deployed.

Our work looks at the feasibility of Non-Stop Routing System Design, where a router can continue to process both control and data messages, even with multiple concurrent software and hardware failures. A thorough discussion on the framework, components, and capabilities is given, and an estimation of overhead, fault recovery and responsiveness is presented. We argue that even though our work is focused in the context of routing, it also applies to any use-case where Non-Stop Real-Time response is required.

# Chapter 1

## INTRODUCTION

Upon initial release, IP architectures were mostly tolerant of intermittent short-lived infrastructure interruptions, largely due to the simplistic use cases of the early Internet Architecture. As those use cases have expanded to include utility services that we depend on, earlier shortcomings of IP architectures have become critically important to address.

We currently use the IP architecture for many critical and utility services such as VOIP (including 911 emergency services), Inter-Institution Banking Transactions, Automobile Automation, Defense Systems, and Industrial Processes (Nuclear Power Plants). These current services impose new high availability requirements upon the underlying infrastructure[37]. From the perspective of the client, the services should be always accessible and need to have a real-time response.

The service ecosystem that is currently deployed in the Internet is much more complex. The complexity is driven by economic requirements imposed upon the systems. Due to its ever-expanding complexity of features and configurations, it takes significantly longer to boot. Thus, restarting the whole system every time that a faulty state is reached is no longer viable, since then the transparency of an always-on system would be broken.

The main objective of this research is to enable components within a service to be more reliable. In order to lower costs and meet new requirements of the overall system, we explore the feasibility of having High Availability (HA) components.

## 1.1 Motivation

Traditional core routing systems[34] in the past have consisted of a single on-board motherboard coupled with a single CPU architecture. As the data traffic has increased, the capacity and throughput of conventional core routers has been stressed to the point that a single CPU architecture is no longer possible. Thus, an ad-hoc practice has developed of interconnecting core routers in a cluster and to use expensive line cards that carry inter cluster traffic rather than customer's data. In observation of this trend, many current core router vendors such as Cisco (CRS) and Juniper (T1600) release platforms that resemble distributed computing clusters[71][74]. Usually, the platform consists of one or more chassis, each composed of multiple line cards that communicate with each other through a high speed backplane switching fabric.

Routing architectures, as presented in [34], have introduced a Single Point of Failure (SPOF). Since a Routing System can be viewed as Safety Critical System (see 3.1), if it is deployed for many operating hours, it is reasonable to assume that any complex component, such as a computer chip or the software within it will fail at some point in time[85].

A necessary requirement emerges to design modular software for HA capabilities that takes advantage of the modern distributed router architecture, enabled by the economic trend of the decrease in price of commodity hardware and increase in performance speed.

## 1.2 First Generation: Single Bus Architecture with Centralized Route Processing

The general design of first generation architecture has a single bus between all the line cards and the CPU. It pretty much resembles a regular commodity PC, where all the packets go through one path. Both the data and the control packets take the same path. The single bus architecture is not desired because it creates a bottleneck[12]. Also the centralized route computation takes too long.

Analogously, a classic forwarding method in Cisco IOS is process switching, such that IP packet forwarding is done by a process that receives CPU control from the scheduler[141][110].

If the process executing on the CPU responsible for examining the routing table fails, then all forwarding stops. Analogously, in a circuit-switched network, once the circuit path is setup, the control logic becomes decoupled from the frame forwarding mechanism. Thus, the control logic can fail and the forwarding will still be able to operate. Thus, from an availability standpoint, circuit-switched networks were better equipped for failure than initial packet-switched networks.

### 1.3 Second Generation: Distributed Route Caching

In second generation architectures, each line card has a route cache that is built up from the routes that are coming into each line card. Single bus architecture is still a bottleneck, however the centralized CPU does not need to be consulted every time, only for control packets and packets which the line card does not know how to process.

Cisco has introduced a method known as fast switching, where the forwarding is carried out at the interrupt level[110]. Fast switching uses a forwarding decision cache. Initially, when the first packet to a destination is process switched, the switch caches its forwarding decision. All subsequent packets to the same destination are switched at the interrupt level using this cache entry. Other Cisco products have SSE (Silicon Switching Engine) has a SP (Switch Processor) co-located with RP (Route Processor)[141]. If the SP has an entry in its cache, it does not contact the RP, however if no cache entry is located, the RP performs process or fast switching.

In addition to the performance improvement benefits during forwarding, the notion of decoupling control path from forwarding data path improves availability, since the control plane can go down and the forwarding will still occur. The method approaches the decoupling technique in circuit-switched networks, and thus shares its benefits.

## 1.4 Third Generation: Switched Backplane with Forwarding Engine

Third generation routers have an architecture where each line card has a Forwarding Engine that hosts a Forwarding Information Base (FIB), a subset of the Routing Table, so that the control plane does not need to be consulted for data packets. Additionally, a single bus plane architecture is replaced by a switched backplane, which is no longer a bottleneck.

The Cisco equivalent is Cisco Express Forwarding (CEF). The need for CEF arose from the disadvantages of fast switching, which populates switching cache on demand, and is not scalable for core routers. CEF uses the FIB table that uses information from the main routing table. FIB always contains all routes and is kept synchronized with the main routing table[110].

## 1.5 Next Generation: Distributed Routers

Modern routing architectures such as Cisco CRS[71][73], Juniper T4000 and T Series Multichassis[74], or Alcatel-Lucent XRS Series[6] resemble a distributed computing platform, where line cards and switching fabric are on separate chassis. The forwarding plane is able to function regardless of whether the routing plane is functional or not, a concept called Non-Stop Forwarding (NSF)[47]. Even though in such configuration it is possible for forwarding tables to become stale and for the incorrect forwarding decision to occur, the fault is not critical since the next-hop router can steer the packet on its correct route.

In order to minimize the downtime of control plane in modern core routers, most Routing Processes (RP) are configured in Active-Standby Configuration[47]. Different High Availability Application Models, including Active-Standby, are discussed in more detail in Section 2.3.2. Even though the switch-over from Active to Standby may be acceptable in most circumstances, it is disruptive to the network since routing tables need to be recomputed and adjacencies reestablished. The approach toward eliminating (or at least minimizing) the switch-over time is called Non-Stop Routing. Industry has tried preventing the disruptions created during the switch-over by adding Graceful

Restart extensions to Routing Protocols, which help to minimize but not to prevent it[47].

## 1.6 Contributions

Our research applies N-Modular Redundancy, fully explored in Section 2.3.3, in the context of Distributed Routing System. The Active-Standby Redundancy Model used in current Distributed Routing platforms is not sufficient to handle the ever-increasing workload demands of customer traffic, both in terms of fault tolerance, Non-Stop performance, and future demands. We propose a Future Internet Architecture, where the Routing Software is able to take advantage of mature technologies used by Distributed and Cloud Architectures in order to increase its performance and fault tolerance. We offer a model of a Distributed Routing System, which can be implemented in a production environment without too much modification. The value it provides to a practitioner is that they are able to evaluate and modify the system before full integration.

## 1.7 Dissertation Organization

Chapter 2 covers key terminology and concepts we will use throughout the rest of the dissertation. Relevant ideas from past research into distributed systems are covered and explained in detail. Chapter 3 looks at past academic works in the past that is related to our research. We either build on concepts from their work or an aspect of their work that is similar to ours. In Chapter 4 we introduce components and mechanisms of Routing and Distributed Systems that we continuously use in our work. Chapter 5 goes over the novel environment and platform that we propose. Chapter 6 explains in detail each component of the actual system and covers the new capabilities of the system. Chapter 7 evaluates our system in terms of performance and overhead. Finally, Chapter 8 provides additional insight into our work, covers our contributions, and presents future work.

## Chapter 2

### BACKGROUND AND KEY CONCEPTS

Since lots of background information was necessary in order to successfully understand our stance in the research literature, we performed a thorough investigation into prior art. In this Chapter, we present key concepts that are relevant to our work and help illustrate how we leverage them. The terminology defined in this Chapter is used throughout the rest of the dissertation.

#### 2.1 Process Groups

A Process/Sibling Group, as defined in [35], is a set of one or more processes which could be on different machines. They usually use group membership framework, which allows them to create, join, and leave groups. A Sibling in this work is referred as a member of a particular process group. Various system implement group communications which allows a sibling to multicast a message to all the other siblings of the process group. Additionally, other systems such as Vsync[138][24] provide a mechanism through which a process can atomically and causally multicast messages to a separate process group (for more discussion on Vsync Framework and Virtual Synchrony, please refer to 3.3.1).

#### 2.2 Scaling

A major challenge that any operations engineer must overcome is the natural and organic expansion of capabilities in a system. For instance, if a product is running on a cloud hosting provider, it would be running on a particular portion of the host's infrastructure, and would consume a certain percentage of the host's memory, compute cycles, and bandwidth. The product is sharing its resources with other users.

The goal of any company is to expand its customers and revenue. When a company decides to scale up in order to deliver its product to more customers, it has two options:

- Vertical Scaling - upgrading the resources a service is capable of using. So, for instance, the business would request more memory, compute cycles, and bandwidth of the machine it is currently using. These resources are not limitless, so eventually the engineers would hit a wall, where they would take over all the memory and compute cycles of the machine, essentially using dedicated hosting.
- Horizontal Scaling - adding more resources. Here, the business, rather than requesting bigger and faster hardware resource, instead requests more machines, and offloads the service to multiple machines. The new machines have the amount of memory, compute cycles, and throughput of the original hosting environment. The interconnection marriage of the multiple machines adds more resources and scalability power to the already existing infrastructure. A major benefit is that the amount of machines a business could interconnect together is virtually limitless, because it is always possible to purchase more machines (within physical limitation of the room they are housed in). However, a major hurdle the engineers face is to designing the service that works seamlessly as a whole among different machines, and the intercommunication latency is not an impediment.

A somewhat similar paradigm shift has occurred in CPU manufacturing, where to expand a computer's computation, it is possible to either purchase faster CPU or add multiple CPUs and parallelize an application.

## **2.3 Different High Availability application models**

The following section attempts to define a High Availability (HA) application model and its subcomponents. Three of the most popular HA models are described and evaluated in detail.

### **2.3.1 Load-Sharing/Stateless**

A load-sharing application is structured in such a way that its state does not reside within the process itself. The state lives externally. The benefit of such a configuration is that the application may crash multiple times and can always easily recover by just loading its state from the external entity. Multiple instances may run

simultaneously and thus provide additional level of availability, allowing a horizontal scaling up of the system.

An example given by authors in [26] is of two programs A and B that interact with a file server and share data through a file which A writes and B reads and modifies. The two programs are written as two components of a single application, and are designed to be run in order: first A, then B. Web servers are usually configured in this way to be stateless. All of the state usually resides in a database that multiple instances of a web front-end interact with. A load balancer dispatches input to be processed by a specific front-end using a predefined algorithm.

Stateless applications do not imply that the programs embody no state, but rather that their states are independent - they share no common states[26]. Stateless applications require a Round Trip Time for each read/write operation, since the state data is stored away from the functions that operate on them[96]. Function shipping paradigm can be used[20] where the data is cached at the application and does not require a RTT on each operation. However, the paradigm enforces sticky sessions[131], where a client is forced into talking to the same front-end application for the duration of its session.

### 2.3.2 Active-Standby

The Active-Standby model has the state in the actual process itself. While the application state is now faster to access and modify, if the process fails, its state is lost forever. Thus, a process group is spawned, where one process among them is designated as the *primary/active*, and all the other processes are designated as the *backups/standby*[56][29]. When a client interacts with the Service, the request is sent only to the *active* process, and only the primary sends a response back to the client. The *backup* processes interact directly only with the *active* process, and external clients cannot communicate with them. The order of operations in active-standby approach is usually as follows:

- A client sends a request to the primary.

- The primary services the request and modifies its state.
- The primary sends a State Update Notification to all the Standby processes.
- Each standby process, upon receipt of the State Update Notification, updates its state and sends an Acknowledgement back to the primary.
- When the primary has received an acknowledgement from each standby process, it sends a response back to the client.

When a primary process dies, a Primary Election Mechanism needs to elect a new *primary* among the *backups*. Different failure scenarios and their recovery mechanisms are illustrated further in [56].

Active-Standby processes need to be explicitly "aware" of one another, thereby forming a distributed state, namely, the respective views that the programs have of membership in the system[26]. The main benefit of active-standby redundancy is that is managed locally. Additionally, only the primary process requires significant processing, where the backups are idle for most of the time. However, the primary-backup approach does not mask all errors in the system. For instance, if the client sends a request and the primary crashes or stalls before it can send a response back, the client must be made aware of the failure. It must be notified of a new *primary*, so that it can retry its request again, but this time with the destination of the new *primary*. Since the client needs to wait until a new primary is elected and then retry its request, the Non-Stop Service transparency is broken (see Section 2.6).

### 2.3.3 N-Modular Redundancy

An application is considered n-modular redundant when the code and state are replicated across more than one process, and executed at the same time, on the same input data. Using this method, it is possible to mask errors, however it is the most resource intensive approach.

N-Modular Redundancy is a very common mode of operation. RAID[109], level 1, mirrors data across multiple drives for redundancy. The earliest known instance of

N-Modular Redundancy was used in the SAPO project during the 1950s. The system implements Triple-Modular Redundancy and uses three parallel arithmetic units and votes on the output[124]. N-Modular Redundancy systems are widely used in Safety-Critical Systems (see Section 3.1). In [24], processes that run in N-Modular Redundancy are referred to as Peer Groups.

More information on N-Modular Redundancy is given in Section 4.2.

## 2.4 Failure Scenarios

The following three situations are indicative of a process misbehaving

- Fail-Stop Failures - A Process exits abnormally. Most of the time, the exit is due to an error from the programmer. However, it may be also because the system cannot allocate appropriate resources (system overload), or because the hardware is faulty.
- Byzantine Failures - The Process is outputting incorrect data. The following class of errors is much harder to detect. The only way is to assert the output against Grounds of Truth that we know must be set in the Output.
- Not Making Forward Progress - A situation where process has not failed however it is not generating output either. The symptom may be a main thread that is hung, either due to some form of deadlock, livelock, or because there are not enough resources available by the system.

Some form of automatic mechanism for failure detection is required for dealing with above scenarios. For example, modern cloud computing environments employ a web server load balancer, which takes on a role of a failure detector and takes a faulty component offline, while signaling the recovery mechanism to be activated[19].

## 2.5 Service Models

There is a fundamental shift in the industry regarding how to design about modern services. In the past, services ran on a specific prebuilt piece of hardware, which when failed, would be nursed back to health by an operator. With the advent of Virtual Machines and Containers, a common paradigm has emerged where Services are now run on generic disposable hosts that can be easily swapped out in the case of

a fault. In [21], the ideological shift in thinking is presented in terms of an analogy of Pets and Cattles, where Pets are prebuilt hardware machines, and Cattles are generic disposable virtual boxes. The popularity of container technology has created a prism through which we see services differently, no longer as monolithic applications, but rather as nimble multi-process abstractions.

### 2.5.1 Let It Crash Model

As the service abstraction is moving towards better decoupling of components, crashes are more likely to occur due to the increase in complexity. Thus, recovery of faulty components needs to be automated. A "Let It Crash Model" is necessary where faults are not treated as outliers but as first-level citizens of the Service Ecosystem[129]. A framework which embraces failures and is used at Microsoft, supports Recovery-Oriented Computing[28].

## 2.6 High Availability

As systems with more complex features are introduced, there are more ways for a particular service to enter a faulty state. Coupled with the ever-increasing growth of commodity PC servers in data centers, the Mean Time To Failure (MTTF) of a single server is once every few hours, either from hardware or software failures in the execution environment[41]. Thus, Non-Stop Services running in data centers that clients can always access and interact with must be designed as Highly Available (HA), using mechanisms quite different from services that do not expect high failure frequency and can periodically become unavailable. The requirements of Non-Stop Services are covered in more detail in Section 4.1.

## 2.7 Consistency

Consistency, defined in [64], is explained in terms of pairs of method calls, where each method invocation matches a response event. Method calls make up many histories of executions, where a history is a set of finite sequence of all possible method invocations and response events.

### 2.7.1 Sequential Consistency

Sequential consistency has a sequential history, which is a subset of all possible histories of executions. The history always starts with a method invocation, and each method invocation is followed by a response event, in the correct program order.

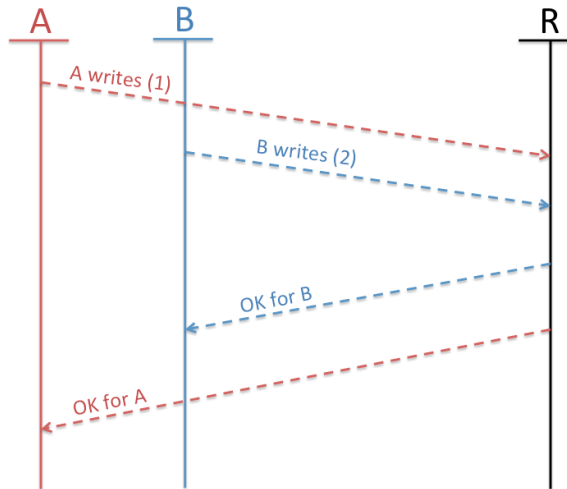


Figure 2.1: No Sequential Consistency

Figure 2.1 shows an example of Host A and Host B concurrently writing to a common register that is resident on Host R, and the requests are buffered with a lock-based queue. Let us assume that the clocks of Host A, Host B and Host R are synchronized. The above example does not implement sequential consistency, since Host A's invocation is not followed by Host A's response event in the correct program order.

In [64], two Principles are stated:

- **Principle 3.3.1** Method calls should appear to happen in a one-at-a-time sequential order
- **Principle 3.4.1** Method calls should appear to take effect in program order

Principle 3.3.1 and Principle 3.4.1 can be interpreted to mean that if there was only one host writing, when it writes a value, it expects the value to follow the sequential logic order of its program. So, for instance, in the Figure 2.1, Host B would expect to see the value 2 when in fact the value 1 is written last. The following example violates

both Principles since the queue is not functioning properly with respect to its method calls. Host A's request should have been serviced before Host B's request. When both properties are satisfied, the correctness property is called sequential consistency.

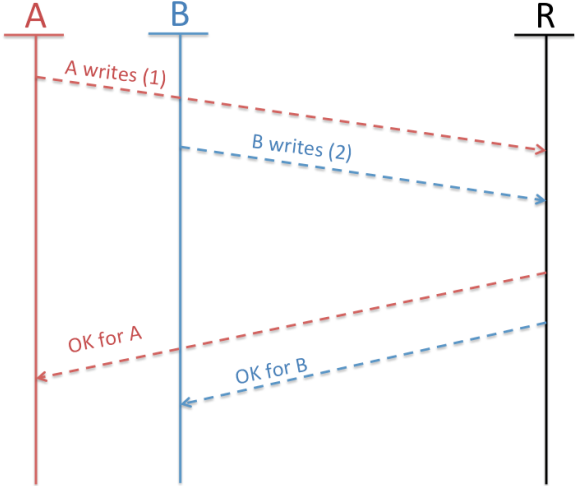


Figure 2.2: Sequential Consistency

Figure 2.2, on the other hand, preserves sequential consistency. There is a method invocation followed by a response event in the correct logic order with respect to the requests that come in. An important distinction to make is that the method execution may happen after the response event, i.e. the response event is only an acknowledgement that the request was correctly received and will be acted upon in the correct program order. In effect, the queue is functioning correctly since it services Host A before servicing Host B.

**2.7.2 Linearizability**

Figure 2.3 illustrates the notion of writing to and reading from two different registers, r1 and r2. Both registers have their own queue, Q1 and Q2, respectfully. Assuming that both registers are initialized with value 0, there could be four possible histories for each host's perspective, H1<sub>a</sub>, H1<sub>b</sub>, H2<sub>a</sub>, and H2<sub>b</sub> (See Table 2.1). The case of H1<sub>a</sub> and H1<sub>b</sub> corresponds to a case when both Q1 and Q2 have the write request enqueued or scheduled for execution, but not yet actually executed. Even though H1<sub>a</sub>

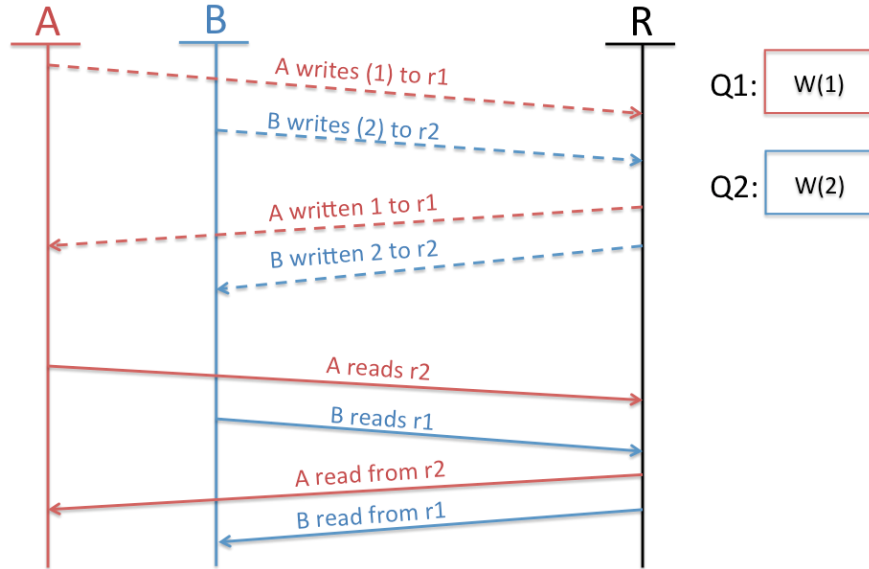


Figure 2.3: Sequential Consistency, Writes and Reads

in itself is sequentially consistent, and  $H1_b$  in itself is sequentially consistent, their composition,  $H1_{ab}$ , shown in Table 2.2, is not sequentially consistent, since it does not satisfy Principle 3.4.1. Therefore, sequentially consistent histories are not composable. In order for their composition to make sense, a stronger notion of consistency is defined, called linearization, where Principle 3.4.1 is changed to

- **Principle 3.5.1** Each method call should appear to take effect instantaneously at some moment between its invocation and response

So the requirement imposed now upon the system is that when a method invocation is received by R, it must be immediately executed before a response can be sent back. Now the response is not an acknowledgement that the method invocation was successfully scheduled, but that it was actually executed successfully. In the period that the execution may happen, no other method invocations must come in and interrupt the execution of the previous operation. Figure 2.4 shows a linearizable system. Now, the subhistories  $H1_a$  and  $H1_b$  are impossible, since the operation is guaranteed to be executed before a response. The only possible history of Host A is  $H2_a$ . Similarly, the only possible history for Host B is  $H2_b$ . Their composition,  $H4_{ab}$  is both

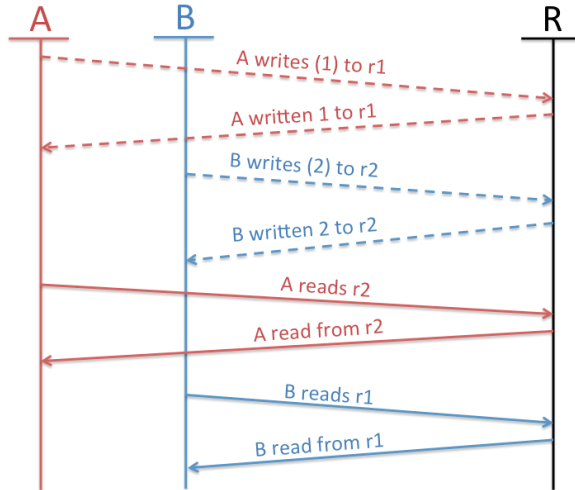


Figure 2.4: Linearization

sequentially consistent and linearizable. This means that a linearizable system is always sequentially consistent, but a sequentially consistent system is not necessarily linearizable.

In our examples, we treat Host R as a single entity in order to present the destination node in terms of abstraction. However, the destination node is usually composed of multiple entities that need to interact among themselves in order to guarantee some form of consistency.

## 2.8 CAP Theorem

The CAP theorem, first mentioned by Brewer[27] states that any networked shared-data system can have at most two of three desirable properties

- Consistency (C) equivalent to having a single up-to-date copy of the data
- High Availability (A) of that data for updates
- Tolerance to network partitions (P)

In order to better understand the concept, I will present the theorem in terms of an example, where there are two redundant shared-data nodes, A and B, as shown in Figure 2.5. If we assume that both network links and processes themselves can fail, and there is a client that is updating both nodes A and B simultaneously, then there

Table 2.1: SubHistory Of Executions

History	Outcome
H1 <sub>a</sub>	A writes(1) to r1, A reads from r2 → 0
H2 <sub>a</sub>	A writes(1) to r1, A reads from r2 → 2
H1 <sub>b</sub>	B writes(2) to r2, B reads from r1 → 0
H2 <sub>b</sub>	B writes(2) to r2, B reads from r1 → 1

Table 2.2: Composition of Histories

History	Effect
H1 <sub>ab</sub>	A writes(1) to r1, B writes(2) to r2 A reads from r2 → 0, B reads from r1 → 0
H2 <sub>ab</sub>	A writes(1) to r1, B writes(2) to r2 A reads from r2 → 0, B reads from r1 → 1
H3 <sub>ab</sub>	A writes(1) to r1, B writes(2) to r2 A reads from r2 → 2, B reads from r1 → 0
H4 <sub>ab</sub>	A writes(1) to r1, B writes(2) to r2 A reads from r2 → 1, B reads from r1 → 2

is a possibility for the link between the client and node B to fail (Case I). In such a case, the data in nodes A and B would diverge. Allowing for this scenario forgoes Consistency in favor of Availability and Partition Tolerance. Availability is retained because other clients can still communicate with either node A or B, even though they get different results. Partition Tolerance is retained as well since A and B have no direct communication link and could be partitioned. If inconsistency is not desired, it is possible to fail node B when a link fault is discovered between B and any of its clients (Case II). Thus all the clients would be able to only communicate with A and always get consistent results. However, high availability of the system would be compromised since the amount of redundant nodes has decreased. Of note is that partition tolerance is also preserved.

The only way to have both Consistency and High Availability in the system is to have nodes A and B share their state via a channel, so that if A's state is updated, the update gets forwarded to B (Case III). However, such a scenario cannot tolerate Partition Tolerance, i.e. the possibility for the link between A and B to fail.

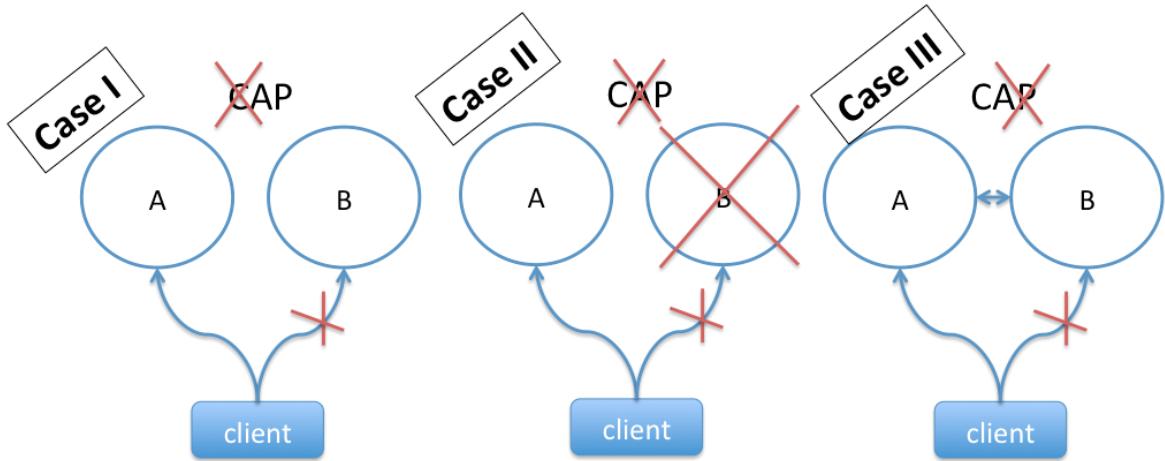


Figure 2.5: CAP Theorem

In this work, I use the CAP theorem as a tool into understanding the relationship between Consistency and Availability. However, the concept is a bit misleading. Partitions in redundant systems rarely happen, so when the system is not partitioned, there is no reason to forfeit Consistency or Availability. Instead, the Brewer in [27] suggests that when a partition occurs, the system should go into an explicit "partitioned" mode with limited operations, and enter recovery. The granularity at which the system may switch its desirable properties is also misleading. The switch may happen at any time in the system's run, even between multiple requests.

The essence of the theorem tries to answer the question about what happens when an error or a time-out happens. Systems that cancel the operation to not risk inconsistency are Strongly Consistent, whereas systems that let the operation go through in order to have High Availability are Weakly Consistent.

## 2.9 Strongly Consistent Systems

Traditional redundant database systems fall into the category of Strongly Consistent Systems. They implement linearization, discussed in Section 2.7.2. The general notion is that the redundant siblings are transparent to the client of the system, and the client treats them as a single entity. If the transparency is somehow broken, due to a timeout or an error, the whole system fails[131].

Replicas of a strongly consistent systems need to have agreement on the next state to transition into. The agreement is known as Consensus[89]. If Consensus is not reached, the request is retried. If Consensus cannot be achieved, the system is stuck in a livelock, essentially failing the whole system. For more information on Consensus and its various system implementations, please see Section 3.2.

## 2.10 Weakly/Eventually Consistent System

Weakly Consistent Systems are the opposite of Strongly Consistent Systems, where no guarantees are made that immediately after an update, the client is able to see the change. The period between the change being received and the update being visible to the client is known as the inconsistency window.

Eventually Consistent Systems are a specific case of Weakly Consistent Systems, where a guarantee is made that if no new updates occur after the last one, eventually all clients will be able to see the last updated value. If there are no failures during the propagation of the update, an upper bound can be derived on the propagation time. Eventually Consistent Systems implement sequential consistency, discussed in Section 2.7.1.

Many modern NoSQL Distributed Hash Table implementations such as Riak[126] or Cassandra[86] claim to be Eventually Consistent. Another highly used Eventually Consistent System is DNS[116]. For more information related to DHTs, please refer to 3.3.2.

A major benefit of Weakly Consistent Systems are that they provide for non-limiting horizontal scale-out[13]. The communication overhead between redundant siblings in Strongly Consistent Systems becomes a bottleneck when attempting to add more nodes to the pre-existing system, i.e. horizontal scaling out[15][117]. Weakly Consistent Systems do not suffer from this constraint. For more information about scaling please refer to Section 2.2.

## Chapter 3

### PRIOR WORK

In order to investigate the prior art of fault-tolerance and resilient routing architectures, we needed to perform extensive exploration into the background of each relevant topic. The following section serves as a gap analysis of current fault-tolerant systems with respect to our proposed one. It is important to note that not all prior literature could be included in this Chapter, however we attempt to include as many relevant pieces of research as possible. Only recent advances have been cited, unless a particular piece of research was seminal to future important advances in the field of fault tolerant architectures.

This Chapter is structured as follows. Section 3.1 describes the fault-tolerance introduced in Safety-Critical Systems such as Automotive, Aerospace and Spacecraft, as a direct result of introducing X-by-wire into its components. Section 3.2 gives a brief overview of the Problem of Consensus in a distributed system and a historical perspective of some of the motivations for Paxos Protocol. Section 3.3.2 defines Distributed Hash Tables (DHTs), and describes how the hashing is distributed to multiple nodes. Additionally, the Section introduces modern DHTs which, through configurable parameters, allows an operator to vary the amount of consistency and availability required by the system. Finally, Section 3.3.2 concludes with some results that illustrate the latency and consistency characteristics of each DHT configuration. We borrow many ideas from Software Defined Networks (SDNs), which are described in Section 3.4. However, SDN Control Architectures are not specifically designed for redundancy, even though the architecture allows for it. Section 3.4 presents an architecture where SDNs are created redundantly. Section 3.5 describes past efforts attempted by various researchers to implement more efficient software architectures on top of preexisting

core router hardware architectures. Section 3.5 concludes with a summary of all the prior architectures and contrasts them with our work.

### 3.1 Safety-Critical Systems

When the failure of a system may result in loss of human life or injury, the system is called Safety-Critical[85]. Initially, the design of safety-critical components was purely mechanical. Modern technology has replaced some of these mechanical devices with electrical components. For instance, modern Flight Control Systems in aircrafts now use Fly-By-Wire, where the movement of flight controls is converted to electrical signals by a computer system that are then sent to actuators. With the movement from manual mechanical controls to electronic-assisted actuator controls, an effort was made to characterize and reduce the source of Single Points of Failure (SPOFs). If the computer system or the signal itself becomes faulty, then the actuators may behave erratically and the plane is likely to crash. In addition to airplane control systems, most modern vehicles have adopted drive-by-wire in the throttle control, braking, and steering. Electronic drive-by-wire allows monitoring of low-level HW components, which provides capabilities such as Electronic Stability Control. The general term for the replacement of mechanical controls with electronic systems is called X-by-wire. The advantages of using electronic sensors rather than mechanical components are that they are cheaper and more reliable than mechanical components. Additionally, monitoring controls can be implemented for vehicle safety and faster response times. However, the issue of fault tolerance plays a major role in these systems since they are safety-critical[30][75].

#### 3.1.1 Automotive Applications

Automotive vehicle safety is mandated by ISO 26262 which, among others[1], enforces explicit safety analysis[134] to derive safety requirements in order to implement hardware and software based fault detection and control mechanisms[83]. The

safety requirements are now much harder to meet due to the increasing amount of new functions<sup>1</sup> which results in higher probability of system failure[83].

The authors in [83] and [16] describe fail-safe and fail-operational systems. A fail-safe scenario occurs when the system can switch to a safe system state in case of detected non-tolerable failures. An analogous model described earlier is the Active-Standby model (see 2.3.2). In fail-safe scenario, the shutdown and switchover of the component is a tolerable scenario. In contrast, in fail-operational systems, the shutdown of a system component leads to a hazardous state, where the critical safety goals cannot be met anymore. Fail-operational systems, such as the throttle or brake controls, requires a stringent set of requirements that can only be achieved with redundancy. Examples of these systems are 2-out-of-3 decision and Triple Modular Redundant (TMR) Architectures[83][16]. Similarly, the focus of our work is on N-Modular-Redundancy, where a Non-Stop Service must provide High Availability requirements. A failure to provide Non-Stop Service leads to loss of revenue for the customers. A survey paper detailing Electronic Drive-By-Wire systems safety is described in [76].

### 3.1.2 Aerospace and Spacecraft

The aerospace and spacecraft industries have always been at the forefront of reliability, where unmanned fault diagnosis and fault recovery is necessary due to its operational environment. The ever increasing requirements for high levels of aircraft reliability and flight safety have led to a growing demand for fault diagnosis and fault tolerant control systems for aerospace applications[50]. Aircrafts usually implement a Fault Detection and Diagnosis (FDD) component[50][4]. A primary objective of an FDD is early detection of faults and detection of their causes. Once a fault is detected, a reconfiguration mechanism attempts to prevent any hazardous behavior from occurring. Many safety-critical components achieve fault tolerance through redundancy[4].

The Space Shuttle, a pioneer in fly-by-wire digital flight control system, was designed to be reliable. The Space Shuttle represents the first planned operational use of

---

<sup>1</sup> For instance, in autonomous vehicle systems[136]

multiple, internally simplex computers to provide continuous correct system operation in the presence of computer failures[120]. The Space Shuttle ran five identical IBM AP-101 general-purpose computers (GPCs)[100][105]. There are four primary avionics computers running in parallel the same computation, and the fifth computer runs a separate Backup Flight System, implementing a Limp-Home capability. The mode of operation uses comparison of output commands and "voting" on the redundant set of computers, allowing the detection and identification of two flight-critical computer failures[120].

The aerospace industry runs in production environments systems such as SIFT[133]. SIFT (Software Implemented Fault Tolerance) achieves n-modular redundancy through the use of redundant processors and memory modules. An early implementation of n-modular redundancy, the application software is composed of multiple application tasks, where each task runs on a different processor. The fact that a task is executed by several processors is invisible to the application software. An application task interacts with the executive software, which performs monitor tasks such as resource management and error detection. The executive software is also responsible for providing correct input values for each iteration of a task through a voter. The executive software consists of local executive and global executive tasks. Each processor executes its own set of local executive tasks, responsible for error reporting and reconfiguration. Each of these tasks are regarded as a separate task executed on a single processor rather than as a replication of some global task, i.e. the application tasks.

Even though the current software and hardware executing on aeroplanes is different from SIFT, the work is very influential on our overall design of the system and fault-tolerance in general. The work of SIFT directly influenced the Byzantine Generals Problem[88].

FTMP (Fault Tolerant Multi Processor), another early effort, uses Triple Modular Redundancy (TMR) Hybrid Model, where spare elements are placed in a pool so that they can substitute for any element in any of several parallel TMR triads[68]. FTMP works at the hardware level, where all buses, processor, and memory modules

are organized into triads and synchronize among themselves. Multiprocessor architecture is exploited for redundancy. Three active bus lines carry data to a voter, which passes the voted-on data to a receiving module. In addition to TMR, Bus Guardians act as monitors, and are charged with governing the status of their associated modules.

FTMP is designed to have a highly improbable loss of capability, with a total failure rate of less than  $10^{-9}$  failures per hour in a flight up to ten hours[68]. In order to achieve such a high level of availability, the system is logically separated into different Fault-Containment Regions; the input processing, configuration control, processor/cache, memory, I/O ports, clock generator and power supply. Similarly, our system is separated as well into different Fault Containment Regions, which we call Fate Sharing Domains (see 5.7).

### 3.2 Consensus Protocols

A common problem every redundantly running system needs to solve is the problem of Consensus. Informally, consensus means that if there are multiple siblings of a process group, and the process group is running in N-Modular-Redundancy, the siblings need to have consensus at some point on a common value. The common value might be used for synchronization, for instance to agree on a common sequence number, or for leader election. The problem of Consensus can also be transformed to a problem of atomic broadcast[43].

A formal description of the problem of Consensus is in [10]. Mainly, the problem of Consensus must satisfy three properties. If there are a set of sibling nodes and they want to agree on a common proposed value, a solution to the consensus protocol must guarantee

- Termination - For every non-faulty sibling, each sibling eventually decides on a value.
- Agreement - All the non-faulty siblings decide on a common value.
- Validity - The value that all the non-faulty siblings decide on must be some proposed value.

The first protocol to attempt to solve Consensus is known as two-phase commit protocol, described by Jim Grey in [55]. The protocol assumes that a coordinator is present, which any of the siblings may take the responsibility of. The coordinator sends a "Propose" message, which contains the value all siblings should consent on. Once all siblings receive a "Propose" message, they send a Yes/No message back to the coordinator. If they said "Yes", they commit the value to stable storage (so that they can undo/redo the transaction). If the coordinator receives a "No" from any sibling, the coordinator sends an "Abort", asking everyone to undo the transaction. Otherwise, the coordinator sends a "Commit". Upon receipt of the "Commit", each siblings commits the transaction as successful, and sends an acknowledgement back to the coordinator.

Two-Phase Commit Protocol solves the problem of Consensus if there are no node failures, since in this case Termination, Agreement and Validity are satisfied. However, there are issues with the protocol if failures are present. The main problem with two-phase protocol is that it is a blocking protocol[119]. If the coordinator fails at the same time that a sibling has declared itself ready to commit the transaction, then the transaction blocks[112]. A very common scenario is where the coordinator itself is a sibling. There is no upper bound on how long it should take a transaction to commit or abort after it has voted. Thus, when the coordinator fails indefinitely, or keeps failing before it has enough time to recover, Termination is not met.

Three-Phase Commit is a non-blocking protocol[119], and has three phases. The first phase is the same as in Two-Phase Commit, where a "Commit" message follows a "Vote" message. During the first phase, a timeout at a sibling or at coordinator results in an abort of the value proposed. The second phase starts when the coordinator sends a "PrepareCommit" message to all siblings. As soon as a sibling receives a "PrepareCommit", the timeout is changed to commit of the value rather than abort, and an "Ack" is sent back to the Coordinator. If the coordinator does not receive an "Ack" from all the siblings, it sends an "Abort" to all the Siblings, otherwise it sends a "Commit" message to all the siblings. The siblings, upon receiving "Commit",

actually accept the value and send a "CommitAck". Non-faulty siblings are no longer blocking indefinitely, since each entity has an unambiguous action it can execute in each situation. More information about Two-Phase Commit and Three-Phase Commit Protocols can be found in [125].

Three-Phase Commit assumes that there are no network partitions, where some siblings are on different parts of the partition. Additionally, it assumes that the network is synchronous, meaning there is a known bound on message delay in the network. When dealing with geo-spatially replicated services, it is difficult to have a good upper bound on message, or processing delays, ahead of time. Typically, timeouts at the coordinator and the siblings is used as a failure detector to indicate a node is down. However, the node may just be slow. The FLP Impossibility Result [51] states that in asynchronous systems, it is impossible to reach agreement, even if only a single node is subject to process failures. Fortunately, in most situations, just knowing that there is some upper bound is acceptable to detect failures.

Paxos, a well-known protocol by Leslie Lamport[87][130][95], preserves Validity and Agreement described earlier. Termination is mostly preserved as well (some rare instances show that Paxos may not Terminate). Paxos is similar to two-phase commit, however there may be multiple coordinators, sequence numbers may introduce multiple proposed values, and the decision of whether to propose or abort is performed with a majority of votes. When there are no faults, Paxos has the same RTT characteristics as two-phase commit. Paxos, Two-Phase Commit, and Three-Phase Commit Protocols attempt to achieve Consensus on the Input Value before it is accepted by the Service (see Section 2.9). This means that when a client sends a request, a response from the Service is sent back only after all the Siblings reach Consensus on the input value, which may cause a significant delay in response.

While simple in theory, the Paxos Protocol is known for its difficulty in implementation and strange corner cases. A use-case paper on implementing Chubby Service[31] at Google has stated that "while Paxos can be described with a page of pseudocode, the complete implementation contains several thousand lines of C++

code"[32]. Due to its complexity and potential synchronization overheads, the industry is moving toward other less rigid systems. Paxos expects an operator to know ahead of time the number of faults the system will experience. Inflexibility in constraints on system availability may not fit some real-world situations. Regardless, it is currently being used in various Open Source projects such as Zookeeper[70], ConCoord[39] and LibPaxos[93].

Recently, there has been an effort to make Consensus Protocols more accessible. Raft[107], a protocol that is equivalent in functionality to Paxos but is more straightforward to implement, is being used in many systems such as Etcd[111], Consul[62] and OpenDaylight[108].

### 3.3 Weaker Forms of Consistency

Due to the limitations of Strongly Consistent Consensus-Based Systems, various other academic research has attempted to implement weaker consistency guarantees.

#### 3.3.1 Virtual Synchrony

Virtual Synchrony is a distributed process group model, developed in Cornell by Birman et al.[24]. The work extended System V process group model to add capabilities for Siblings to be tolerant of process failure when exchanging state messages. The authors implement **gbcast**, **abcast** and **cbcast**, which are three different multicast implementations with different guarantees. Multicasting may be used for synchronization as described in Section 4.4.3. **Gbcast**, which stands for group broadcast, is in many regards similar to Paxos, and could be transformed into Paxos. **Gbcast** is used for process group membership dissemination[25] (see Section 4.4.2). **Abcast**, which stands for atomic broadcast, is similar to **gbcast**, however it is used for atomic multicast among Siblings of a Process Group. **Cbcast**, or causal broadcast, is a weaker synchronization model of **abcast**. If used correctly, **cbcast** can provide the same ordering guarantees on message delivery as **abcast** at much faster delivery time, as shown by authors in [23]. As opposed to **abcast**, **cbcast** uses causal consistency, a weaker

form of consistency than strong consistency, where only causally related messages are delivered ordered. There are no constraints in **cbcast** on ordering of Concurrent messages. When using **abcast**, its strong guarantees require that siblings synchronize on every message, thus achieving latencies similar to Paxos. However, the weaker causal consistency model of **cbcast** only synchronizes when a primitive *flush* invocation by the operator forces the system to go into a consistent state[137].

For our work, we use linux TCP channels for communication among process groups. Although we currently do not use virtual synchrony, we believe that the communications channel is extensible enough to be easily be replaced by something else, such as a virtual synchrony framework.

### 3.3.2 Distributed Hash Table

Distributed Hash Tables (DHTs) are based off academic research that has great influence and success in the industry. A Distributed Hash Table (DHT) provides a service similar to a hash table, with a key difference being that it can be spread across multiple nodes. It is a form of a decentralized distributed system.

The hash table distributes its keys and values across multiple nodes uniformly. The original idea is based off consistent hashing[80], where a logical ring is formed around all the nodes. Early academic implementations of DHTs such as Chord[122] use consistent hash rings that automatically map hashed keys to nodes. If we assume that there are 8 nodes, A - H, and a keyspace that maps to an integer value in the range 0 - 255 with a hash function  $H$ , then we can make a function that maps each portion of the keyspace to a different node. Figure 3.1 shows a specific example. The output of the hash of key  $K$  is an integer value that maps to Node B. Whatever node a particular key is mapped to, all of its associated data is stored there as well. In order to survive single node failure, the data that is stored in a particular node is replicated to all the nodes following it in the ring. A user-defined parameter  $N$ , called the replication factor, dictates how many nodes the data is replicated across. From Figure 3.1, if the

replication factor is set to three, the data associated with key  $K$  is stored on nodes B, C, and D.

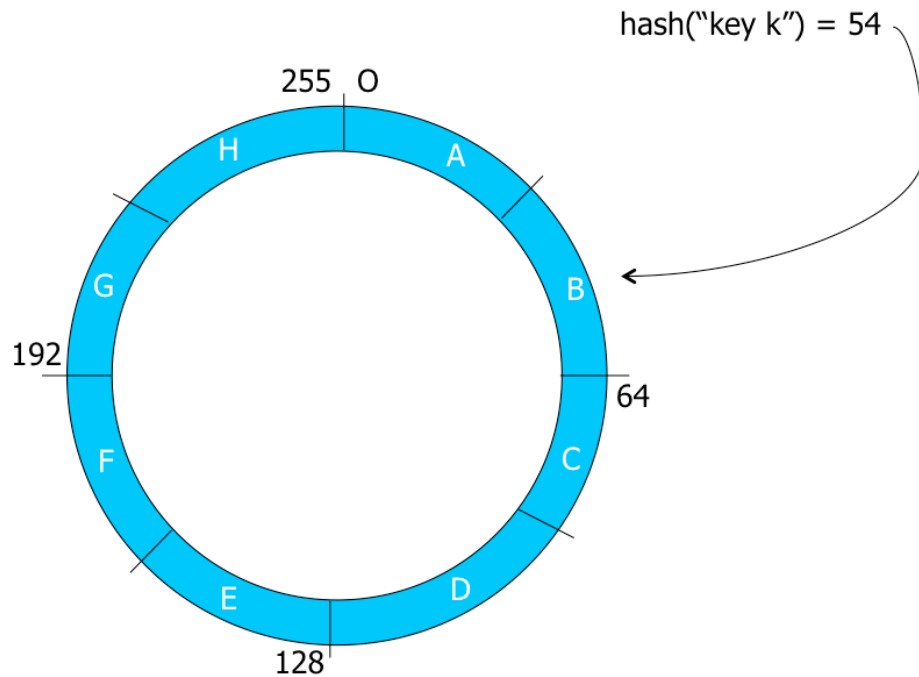


Figure 3.1: Ring with Consistency Hashing

Other systems such as Bayou [127] use eventual consistency (see 2.10) and epidemic algorithms[44] to allow input from clients with no need for blocking or locking. A survey paper on Overlay Network, including DHTs, can be found at [92].

Modern NoSQL DHT implementations, such as Dynamo and its derivatives Cassandra and Riak, are directly influenced by Chord and Bayou. Consistent hashes are used similarly to map client data to specific nodes, and epidemic anti-entropy techniques[113] are used to eventually synchronize data among multiple nodes. In addition to the replication factor  $N$ , DHTs also typically have a user-defined parameter  $R$  and  $W$ , which allow the amount of availability and consistency of the system be configurable.  $R$  defines how many of the  $N$  nodes need to respond to a client when a read occurs.  $W$  defines how many of the  $N$  nodes need to respond to a client when a write occurs.

Nominally, changing  $N$  once the DHT has started is not advisable. A user allocates resources and determines service reliability required ahead of time. It is possible however to change  $W$  and  $R$  on-the-fly for each request. By increasing  $R$  and  $W$ , the amount of consistency in the system is increased, but the turnaround time for calls to complete is decreased. The availability and durability of the system is decreased as well, since if  $M$  nodes fail such that  $W > N - M$  or  $R > N - M$ , the write or read calls will be unable to complete and the system will deadlock. On the other hand, if  $R$  and  $W$  is decreased, then the system becomes tolerant of node failure, however a period of time exists during which the DHT might return an inconsistent result. The inconsistency window, as introduced in Section 2.10, is the time it takes for the modified data to propagate to  $N - M$  nodes.

A DHT operates in n-modular redundancy mode, since data is replicated in more than one node, and the client acts as the voter. DHTs are one of the few modern software systems which have largely achieved reliable operations with no single points of failure (SPOFs). Our system draws inspiration from the NoSQL DHT movement, where we are also running a system in n-modular-redundancy mode and the communication between siblings is only through eventually consistent communications. This enables a sibling to respond to requests as fast as possible and not wait until all the siblings are synchronized. For more information about our system and its details on Consistency, please refer to Section 6.2.4.

Figure 3.2 and 3.3 presents an experiment where 3 nodes are running Riak, and are used by high-level applications. A high-level application performs two operations, **get** and **update**, respectively. In Figure 3.2 we set  $R$  and  $W$  equal to One, and thus a response returns faster. In Figure 3.3 we set  $R$  and  $W$  equal to All, meaning that all 3 nodes need to respond to the client. This guarantees that the response is consistent. The Throughput graph on both figures represents the overall throughput of all operations, and displays how many successful and unsuccessful errors were sent back to the client. The throughput when  $R, W$  equal All is much higher than  $R, W$  equal One, since there are two more messages in All case for each message in One

case. However, from the Latency Graph we can see that the latency in All case is much higher, the maximum being about 4000 ms, whereas it is 3000 ms in One case. Additionally, the Throughput Graph indicates that errors have occurred in the All case. Specifically, there are 5 errors on **get** and 13 errors on **update**. However, the One case has no errors, since the high-level application needs only a response only from its local node and does not need to synchronize with the other ones.

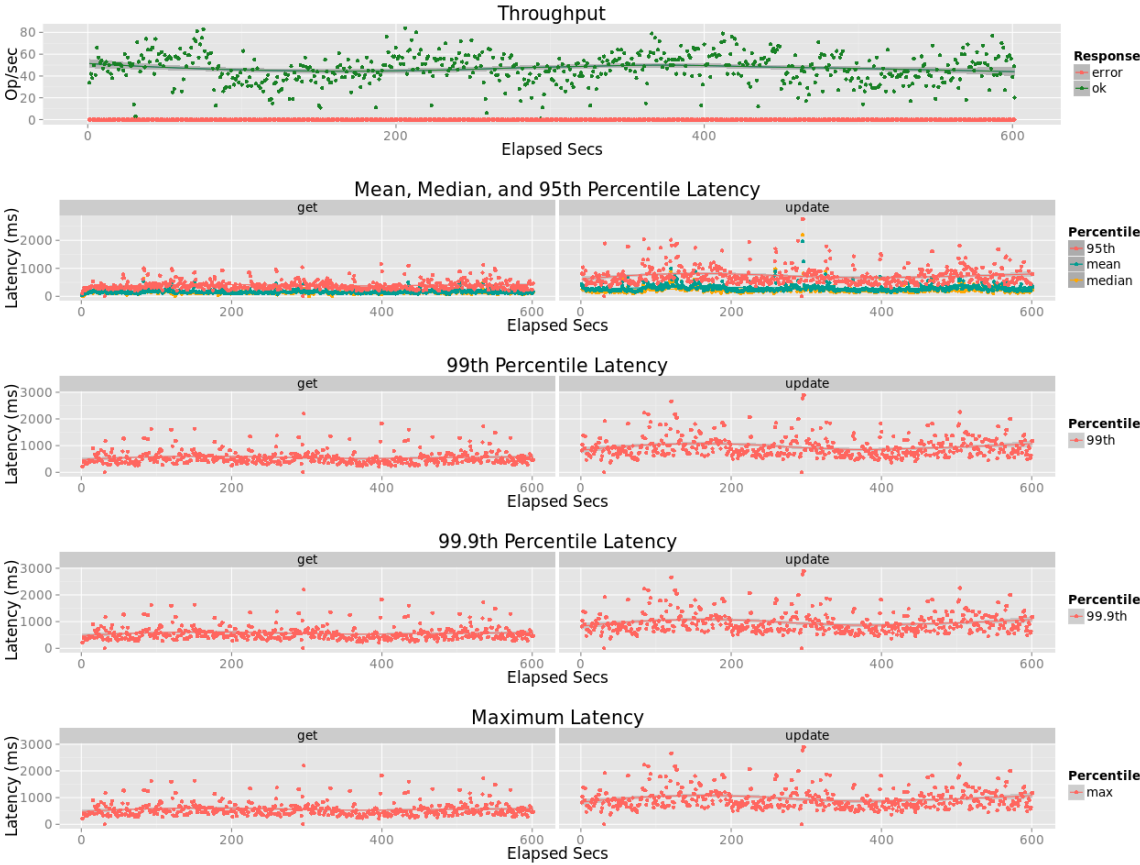


Figure 3.2: R and W = 1

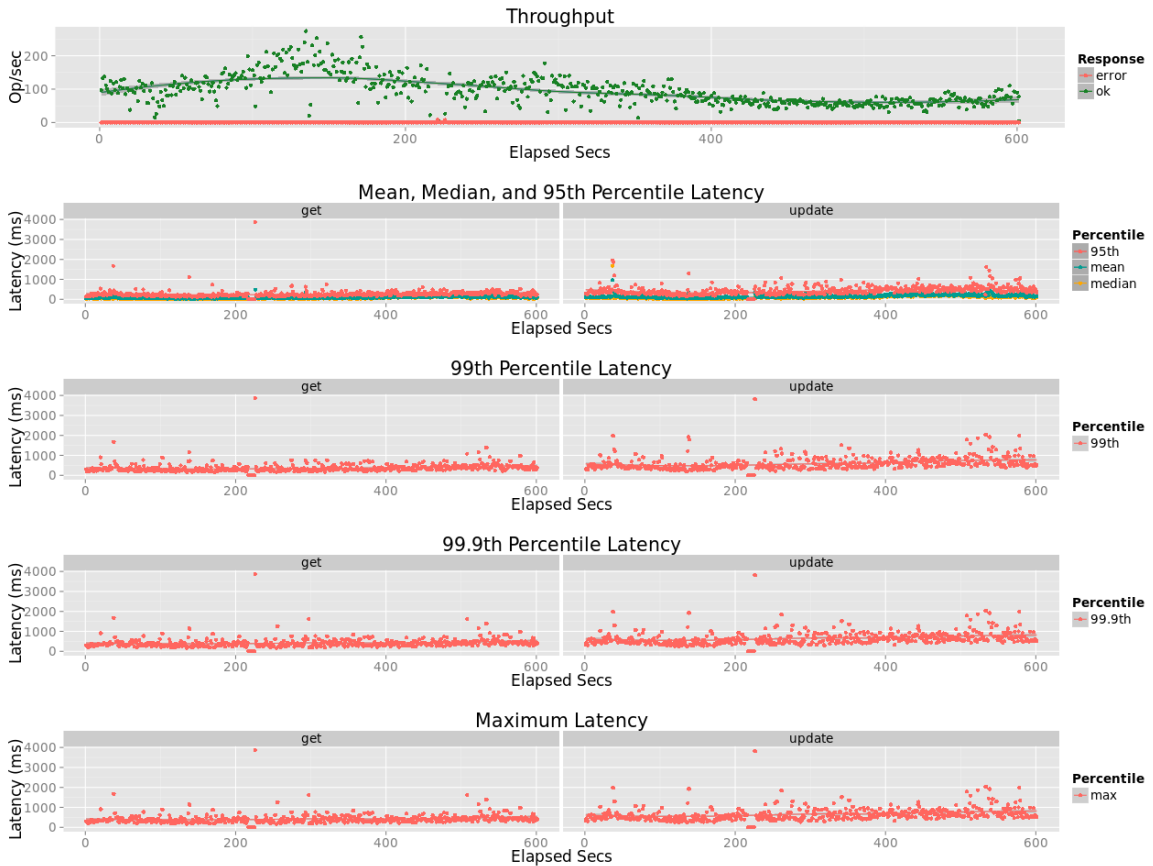


Figure 3.3: R and W = ALL

### 3.4 Software Defined Networks

Software-Defined Networking (SDN) is an effort to decouple the Data Plane from the Control Plane and to logically centralize network intelligence and state[52]. The push is to physically separate the Data Plane from the Control Plane, and to outsource the more complex policy processing logic into the Control Plane. Various previous efforts have attempted to separate the forwarding and control components in Routing Architectures, but for various reasons, have not gained traction[49].

The Data Plane is usually implemented in hardware, in order for the forwarding to happen as fast as possible and to prevent software bugs. However, a Data Plane implemented in hardware is inflexible and usually custom-built. A natural evolution of SDN effort led to a need for an open protocol specification between the Data Plane and

the Control Plane, which all hardware manufacturers must support. The most widely used protocol is OpenFlow[53]. Usually management protocols such as SNMP[77] exposes only high-level capabilities specific to an application. OpenFlow, on the other hand, requires that switching functionality at much lower level is exposed[128]. OpenFlow allows for dynamic reprogramming of the hardware forwarding path.

When the Control Plane resides on the same device as the Data Plane, the Control Plane is usually logically paired with a single Data Plane. For example, the Control Plane of a given Router cannot directly control the Data Plane of an adjacent Router. If the Control Plane is decoupled from the Data Plane, a logical centralization of all Control Planes is possible. Logical centralization only means that the Control Plane is able to control any network appliance that implements packet forwarding at the Data Plane. Physical centralization is undesired since then it becomes a Single Point of Failure.

Due to the amorphous nature of SDNs, the amount of research in that area has exponentially increased. Not a lot of focus has been given to the reliability benefits that SDNs can provide. According to [48], there have been solutions for BGP implementation for SDN networks, however there are very few studies on scalability and high availability of BGP. The work of OFBGP[48], an implementation of a BGP Architecture for SDN, looks at scalability and high availability.

In order to handle a growing amount of BGP neighbors and IP prefixes, the authors of [48] split a BGP process into BGP Protocol and BGP Decision modules. A separate BGP Protocol process is started up for each distinct BGP neighbor, thus the processing of Routing messages does not increase with the increase of BGP neighbors. A distinct BGP Decision process works on the IP prefix tree that it is responsible for. Essentially, the way the system works is that TCPR[123], a reliable TCP framework, is used to communicate between a BGP Neighbor and an individual BGP Protocol. The BGP Protocol process checkpoints ingress messages to Cassandra[86], a Distributed Storage Service. A Communications Service such as ZeroMQ is used between BGP Protocol and BGP Decision. An instance of the BGP Decision Process Group computes

route selection, calculates the best route, updates FIB and installs flow entries onto OpenFlow switches. In order to achieve high availability, the BGP Decision process periodically snapshots its internal state. It is critical to find the correct granularity of when to checkpoint internal state snapshots. When a process crashes, its state can be restored with a combination of the internal state snapshots and the ingress messages from BGP Protocol checkpointed earlier.

OFBGP uses ZooKeeper[70] as a process registry (see Section 4.4.2), where each task of BGP Protocol and BGP Decision registers upon startup. Each task is able to see the liveness status of other tasks. In the case of node failure, other computing tasks are notified and the work is migrated to different nodes of the cluster.

Not unlike the work presented in [48], our work leverages the benefits of SDN architecture in order to exploit possible redundancy capabilities. SDNs are not built with redundancy originally, and both our work and [48] focus on this aspect. The work of [48] has an original way of performing checkpointing, which we have considered implementing in the future (see Section 8.3). The benefit of such a checkpointing solution is the decreased process restart time and potentially smaller message footprint size. However, the system only implements load-balancing of different BGP neighbors and different IP prefix trees. For a more thorough discussion contrasting our work to that of SDNs, please refer to 5.2.1.

### 3.5 Next Generation Routing Architectures

Research related to New Generation Routing Architectures proposes new ways of looking at Routing Systems. As mentioned in Section 1.5, modern routing architectures look less like a monolithic appliance and more like a distributed modular computing platform. Using this new computing model brings some exciting opportunities in software architecture that would not have been possible otherwise. Authors of [57][38] believe that these new routing platforms improve scalability and availability. Scalability and flexibility is improved because modules can be added dynamically

on-demand, as capacity requirements increase. Availability is improved by adding redundancy and replication over multiple modules, and faults can be isolated to specific regions.

The work of Hidell[57][65][66] discusses an implementation of a distributed router using off-the-shelf UNIX platforms for the Control Elements (CEs), and specific mission-specific hardware platform for the Forwarding Elements (FEs). A protocol, called Forz[57], is used for communication between both CEs and FEs. Forz encapsulates Netlink messages, which FEs use to modify the Forwarding Plane on behalf of the CEs.

Our initial design goal, similar to Hidell’s work, is to use COTS architecture (see Section 5.2). We also separate the Control Plane from the Forwarding Plane so that the Forwarding Plane can be substituted based on necessary requirements (see 5.4). We implement a communications protocol between the Control Plane and Forwarding Plane (see 5.2.1), and utilize Netlink messages from Control Plane to affect the Forwarding Plane. Communication at the Control Plane is performed via SIS-IS sockets(see 5.6). Hidell’s work instead disseminates information among the CEs through a nack-based multicast protocol called NORM[65].

The work of Hidell does not consider the issues associated with synchronization among multiple CEs, no supervisors are implemented, and their implementation with regards to redundancy is limited. The authors implement a version of BGP, where the range of IP address calculation is evenly shared among multiple CEs, and each active CE also acts as a backup for its neighboring CE. This work does not consider n-modular redundancy in its implementation, and in its experiments does not consider recovery from failure. Our work, discussed in detail in Section 5, considers in detail the issues of consistency among multiple processes, and examines state transition during process failure.

The authors of [103] argue that the software has not yet caught up to the hardware of core routers. The reason is that routers are designed with only one powerful

controller card, and throughput is increased only if more chassis are added with additional control cards. The authors propose an innovative solution that is based on task sharing between control and line cards. A Route Table Manager (RTM) generally runs as a single process on an individual Control Card. The authors propose splitting the RTM and running multiple copies of it on separate Line Cards.

Nguyen's work[104] presents an architecture where functional distribution of the control plane enables the routing and signaling functions to be shared between control and line cards. The distribution of the system is presented in terms of two routing protocol case studies: OSPF and BGP.

A distributed OSPF architecture consists of OSPF Control Component (OCC), which operates at the control card level, and OSPF Link Component (OLC), which operates at the line card level. Responsibilities of OCC are to maintain a global view of the network topology, update best routes to RTM, and interact with OLC in all the line cards. An OLC runs OSPF Hello protocol, synchronizes its Link-State Database with other OLCs and floods Link State Advertisements to the external world. The OLC module runs on each line card that is connected to an OSPF area. This way, the OLC reduces the burden of the central control OCC. Authors claim that the architecture is more scalable since the router resources used by this protocol could be adapted to the amount of the routing traffic of the network where the router is deployed. Additionally, if one line card goes down, the adjacencies with the neighbors on other line cards is not lost. This helps improve fault tolerance and availability of the router.

When a BGP protocol is distributed, it becomes more scalable, and enables the handling of more BGP sessions with its peers. The learned information from each peer is distributed on different control cards, so that a control card is responsible for a limited number of peers and their route information. Each BGP control process computes a Local Version of the RTM, that it then sends to a Global RTM builder that assembles all the Local RTMs and updates the local RTMs of the other BGP control processes. The authors suggest an architecture, where resiliency is achieved by designing a fault tolerant protocol using a primary/backup scheme where in case of a

failure of one process, a switch-over to the backup takes place so that the BGP sessions are maintained.

Nguyen’s work is broader in scope than Hidell’s however it is less complete. Unlike Hidell’s work which uses COTS hardware, Nguyen’s work is implemented on Hyperchip hardware, which most researchers do not have access to. In both case studies of distributing OSPF and BGP, the work of Nguyen mostly deals with load-sharing routing tables among peers or address ranges, and in limited scope looks at distribution in order to achieve fault-tolerance. The issues of synchronization among multiple line cards is not addressed as well.

The work of Zhang presents SDBGP[140], where the BGP protocol packet processing and route computation is distributed. A central routing process is split up into distinct parts so that each Agent is only responsible for route processing and calculations from an individual neighbor. It is similar to Nguyen’s work in regards to the distribution of processing onto multiple control nodes. A key difference is that there is no centralized single entity that agents communicate with. SDBGP assumes node failure in its design, and therefore does not introduce any centralized synchronization points. Instead, the agents communicate among themselves using reliable UDP multicast, and have a workload scheduler which acts as a monitor. Similarly, our work does not have centralized architecture, and implements monitors at the siblings. In SDBGP, only the agent with the lowest identifier runs the workload scheduler. The workload scheduler in all other agents is on standby. If the scheduler agent fails, the next agent with the lowest identifier starts its scheduler. Our leader election mechanism is slightly more involved, though achieves a similar goal of electing one entity which has slightly elevated responsibilities from its siblings.

The work of Hamzeh[58] explores the issue of keeping multiple RIBs in a distributed routing architecture consistent. This work looks beyond distributed partitioned processing of IP ranges to an architecture where multiple Routing Information Managers (RIMs) process different messages that belong to the same IP range. For instance, if the messages  $r_i^+$ ,  $s_i^+$ ,  $r_i^-$  arrive at an input queue of ingress port, and there

are three RIMs, i.e.  $RIM_1$ ,  $RIM_2$ ,  $RIM_3$ , each RIM may process a different message.  $RIM_1$  may process  $r_i^+$ ,  $RIM_2$  may process  $s_i^+$ , and  $RIM_3$  may process  $r_i^-$ . After processing these messages, the RIMs synchronize. However if they only synchronize with exchange of messages by using "consistency checker" (see Section 4.3.4 and [132]) that commercial routers use to compare consistency between the RIB and FIB, the three RIMs will have different view of the final RIB. Authors of [58] present a causal consistency model designed specifically for BGP. Causally related messages are distinguished from concurrent messages which are not related in any way. A partial message ordering at each RIM uses Logical Clocks in order to guarantee causal consistency.

An implementation of a fault-tolerant message-logging based architecture is presented by Hamzeh in [59]. Using message logging, RIMs checkpoint their messages in case they need to be replayed later on. Heartbeat techniques are used for failure detection. A RIM can function both as primary and backup. A ring topology guarantees that each primary has a backup in its clock-wise direction. The authors address the issue of consistency during recovery, stating that previous fault-free causal consistency techniques during normal operations does not lead to the desired levels of consistency. A modified message replay algorithm is presented in [59].

Our architecture similarly uses checkpointing for recovery and heartbeats for failure detection. Our work has a different model for execution, where n-modular redundancy rather than primary-backup approach is considered. We use eventual consistency for synchronization and consistency between replicas. However, the consistency in our system is less fine-grained and uses a distributed key-value store as a substrate. Even though we detect RIB divergence at the voter, the approach taken by Hamzeh is of interest to us since we could apply their work toward the effort presented in Section 4.3.2.

The work of Eric Keller in building a Bug-Tolerant Router[81] uses n-modular-redundancy to implement Software and Data Diversity (SDD) transparently using different implementations of Routing Protocols. Different versions of XORP[60] and Quagga[78] run concurrently. N-Version Programming[11] is used, which suggests that

the concurrent execution of multiple versions of the same specification greatly reduces the probability of identical software faults from occurring in two or more versions of the program.

In addition to running multiple software versions, Keller reports that varying the execution environment also masks some bugs that are time-related. Artificial delays are introduced to alter the timing and ordering of updates without affecting the correctness of the router. A source of common bugs comes from the timing and ordering of events that occur from the order of connection arrivals or status changes in network interfaces. The author of [81] avoids common bugs by diversifying the connections to RPs, and randomly delaying and restarting connections for certain instances.

A group of RPs is made to run transparently through the use of a hypervisor. The hypervisor is composed of a replicator, a FIB voter, and an update voter. The replicator acts as a coordinator and sends a copy of all received data to all router instances. The update voter determines the external messages that are to be sent to the peer router, and the FIB voter determines the updates that should be sent to the FIB via Netlink messages. When a routing instance needs to be restarted, either because it crashed or is outputting buggy results, the hypervisor bootstraps the new instance. To assist with bootstrapping, the hypervisor keeps the last update for each prefix, and replays it for new instance upon startup.

The Bug-Tolerant Router implements three different schemes of voting. The wait-for-consensus scheme requires that all inputs are either received, or a timeout occurs for a particular input. The scheme guarantees that up to  $k$  faulty processes can be masked if there are  $2k + 1$  instances running. The scheme is the slowest to react to events since all inputs must be waited on. The master-slave approach elects a certain process as master, and makes all the other processes slaves, so that they only cross-check the results of the master. A master may temporarily output erroneous information, in which case the master is taken offline and a slave is promoted to master. When slave overthrows the master, the voter readvertises any changes between the slave's and master's routing tables. While the scheme is about as fast as the base case where there

is only a single process executing, the scheme is most prone to outputting incorrect data which later needs to be corrected. A compromise is to use continuous-majority voting scheme, where voting is performed every time a process sends an update, and updates are only sent when the majority result changes. As a compromise, the scheme guarantees faster reaction to failures than master-slave, and faster response than wait-for-consensus since the majority result may be reached before the slowest instance finishes computing. However, there may be more multiple erroneous and corrective updates, since the voter may be working with erroneous data.

Both our work and Eric Keller’s work attempts to mask and tolerate faults in a transparent fashion. However, there are multiple distinctions in the architecture. We do not keep all system monitoring and management tasks in a single location, such as hypervisor. Rather we diversify their locations as much as possible. We keep the bootstrapping information in a distributed redundant data store, thus eliminating SPOFs. Our voting strategies are inspired by Eric Keller’s work and are explained in detail in Section 4.2.2. Our voting strategies are further analyzed in Section 7.4. The architecture assumed in Eric Keller’s work is of a first-generation routing system, and thus does not address problems such as process migration or state synchronization.

Table 3.1: Table of Prior Work

<b>Author</b>	<b>Novelty and Relation to our Work</b>
Hidell	Uses x86 architecture; Implements primary/backup
Nguyen	Uses proprietary architecture; Local routing tables synchronize with Global routing table; implements primary/backup
Zhang	No Global Routing Table; Local Tables Synchronize Among Themselves; Workload scheduler in each local process
Hamzeh	Routing Table Synchronization; Message Logging; Heartbeat
Keller	N-Version Programming; Transparent RP through a hypervisor; Different Voting Schemes

Table 3.1 provides a brief summary of the novelty of each work and its relation to our work. The work of Keller is most closely related to ours, and can be used as a base implementation for future work described in Section 8.3.

## Chapter 4

### BUILDING BLOCKS

A Highly Available Reliable Routing System is made up of multiple building blocks, that compose together to form a synergetic system that is able to meet requirements previous systems have not been able to handle. This Chapter begins by listing some of the Requirements for the system. A method of splitting and voting with N-Modular Redundancy is shown. The Routing Table Structure, and its relation to redundancy and synchronization, is analyzed. Classic techniques present in all HA systems, such as checkpointing, process group membership (resource registry), and state synchronization, are described. The Chapter concludes with a focus on an Distributed Key-Value Stores, which can be used as substrate to implement HA techniques described earlier.

#### 4.1 Requirements

The key requirements for a Resilient Redundant Routing Framework are

- High Availability (non-stop) device - an N-Modular Redundant distributed system composed of processes that implement well-known routing protocols with checkpointing and recovery.
- Real-time response - Canonically, the operational requirements in the ISP industry are that the maximum processing time to make a decision for a received packet should be no more than 200 msec with no more than 10 non real-time events per day<sup>1</sup>. Therefore, the siblings must react to messages in real-time, with as little latency as possible between them.

---

<sup>1</sup> These requirements were based off internal presentations and discussions with various industry experts in Internet Architectures

- Auto-discovery and self-configuration - Processes in a sibling group should be able to discover the status of the other siblings in a straight-forward and consistent manner, and configure themselves accordingly. The auto-discovery and self-configuration mechanisms should be triggered upon process start-up and adapt as components change. In general this same philosophy on auto-discovery and self-configuration should be used across all components found in the routing system.
- Non-stop software upgrades and downgrades - As outages have significant impact, it is desirable to enforce a requirement allowing the system to not be taken offline for maintenance reasons, such as reinstalling software. This requirement enforces the design of a high-availability router. The mechanisms allowing non-stop upgrades and downgrades should also allow multiple versions of a software component to co-exist, thus allowing online testing of different software versions. Only one version would be active with respect to the overall system. Other versions would only be observed for their proper operation.
- Process Migration - Due to the non-stop capabilities of the system, a service should be able to survive single process or single line card failure, where a process and its state can be migrated to an alternate line card on the system.

## 4.2 Splitting and Voting with N-Modular Redundancy

A N-Modular Framework is realized through the help of a splitter and a voter. A splitter is a module which receives a single input and sends it out to the n-redundant processes. The voter module must vote on multiple inputs and generate output[90] These splitting and voting modules are added at the boundaries of all the processes that are made redundant, so that they seamlessly integrate with the rest of the system.

Figure 4.1 presents multiple Process Groups, where each Process Group is a Message Propagation Stage. On each Sibling in the Process Group, a voter checks all ingress messages for correctness. A sanity check filters out all obviously bad inputs, such as inputs with bad timestamps, or negative sequence numbers. The voter, explained in more detail in the following section, either outputs True or False, based on how the voting is performed. If the output is True, the rest of the body in the sibling process is executed.

The following subsections look at different ways of implementing a voter and various strategies for deciding whether an output is correct or not. This section concludes with a look at the case when there is no agreement.

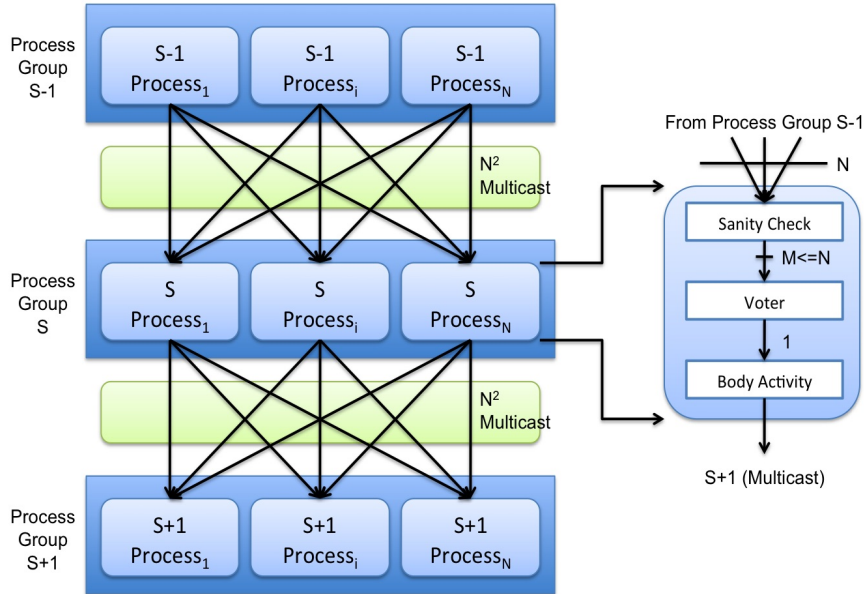


Figure 4.1: Processing Stages

#### 4.2.1 Voting Implementations

Different voting implementations may behave in different ways. A vote is a comparison between all input messages. If there is commonality between the input messages, where the commonality is predefined by the implementation, the input is marked as correct and the input messages are collapsed into a single input message that is then sent to a downstream process. If no commonality is found, then a fault has occurred in the system and a fault recovery mechanism is triggered (Section 4.2.3).

Voting implementations may vote on input by

- All or Nothing Message Comparison - In order for the vote to succeed, all messages being voted on must be equal. This is the strictest form of comparison. A byzantine failure will prevent the system from functioning since no vote will be achieved with a single erroneous message present.
- Majority Message Comparison[90] - More than 1/2 of the input messages must be equal. When a majority is reached, the messages that are in majority are

marked as correct. The comparison allows for masking of byzantine failures, as long as they are present in the minority of the messages. An even number of input messages may result in situations where a tie is reached, thus preventing the voter from distinguishing correct from incorrect inputs. To prevent such situations, siblings are usually deployed in odd number. The disadvantage of majority comparison is that agreement may happen on incorrect input. If the amount of incorrect input messages are greater than the amount of correct messages, the incorrect input will be chosen. The algorithm is presented in more detail in Algorithm 1 and Figure 4.2. The *inputQueues* are the queues that are created for each sibling that stores its ingress messages. In Figure 4.2, messages P, X, and Z arrive and don't match, and are therefore buffered; after which messages Q, Y, and S arrive and are buffered as well. When message Q arrives at queue Q1, a majority match occurs. Message Q is passed to next stage, and all the previously received messages are flushed. The voting implementation may either flush all its input queues, or it may flush only queues Q0 and Q1, since only they participate in voting.

---

**Algorithm 1** Voting on Majority

---

```

1: procedure VOTE(incomingMsg)
2:   while incomingMsg m not in inputQueues do
3:     add m to appropriate inputQueue
4:     if majority in inputQueues then
5:       pass matched packet p to next stage
6:       flush any previously received messages that matched majority from [all
input queues] or [input queues that participate in majority]
7:     end if
8:   end while
9: end procedure

```

---

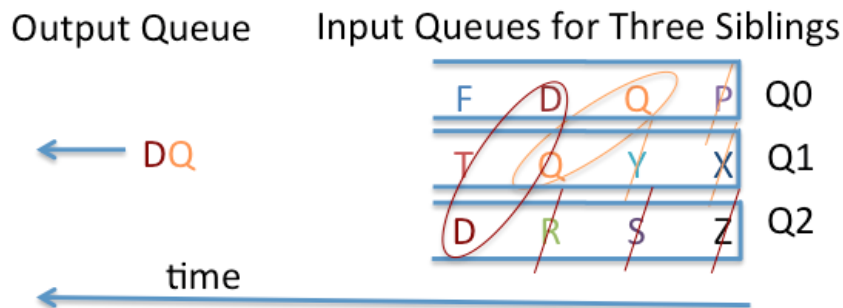


Figure 4.2: Voter on Three Sibling Streams

- Edit Distance - An edit distance[101] is the number of operations necessary to transform one input message into another. If operations are limited to removal

or insertion of a single character, or the substitution of one character for another, that particular form of distance is called the Levenshtein Distance[101]. Levenshtein Distance closely models situations which may occur in byzantine faults, where a bit may be flipped, accidentally added, or discarded. A voter may find the number of operations required to transform one input message into another, and if the number of operations required for transformation is less than a given threshold, then the input messages are marked as correct. However, in such a situation, it is harder to decide which single message to collapse all inputs to. In all scenarios, it is best to choose the input message which has the smallest edit distance to all other messages.

- Context-Aware Comparison - A voter that is aware of input message format may parse the input messages and extract information from it. Using the extracted information the voter is able to perform Cluster Analysis, use Support Vector Machines or other Machine Learning methods. The context-aware comparison can be operator defined, where the operator of the system determines how to perform comparison between input messages.

#### 4.2.2 Different Voting Strategies

We denote the amount of siblings desired by the system as  $N$ . The parameter  $N$  is a configuration parameter and is set by the operator before initial operation. Since siblings may fail during operation, there is a number of siblings currently live and operational, which we call  $n$ . At any given time, the assertion  $n \leq N$  holds. Voting strategies dictate when input should be voted on:

- ANY Voting Strategy - ANY voting strategy requires that we receive input messages from at least  $n/2 + 1$  siblings in order to vote on the input. Since we are voting on only live siblings, no waiting is necessary on dead siblings which need to be restarted. However, a byzantine failure might not be detected when at least  $n/2+1$  siblings output faulty inputs which would be voted on as correct, and when  $n/2 + 1 < N/2$ . An example is shown in Figure 4.3. If there are 7 siblings, and 2 of them are dead, then the number of live siblings,  $n$ , is 5. The received erroneous messages are  $n/2 + 1$ , and they could all have the same fault, thus the voter would consider them correct when they should not be.
- ALL Voting Strategy - ALL voting strategy requires that  $N/2 + 1$  siblings send a message before the input can be voted on. While more robust against Byzantine failures, the voting strategy can result in a deadlock, where the necessary siblings are never able to restart fast enough to send a message. For instance, in Figure 4.3, the voter will wait on one more message to be received before it could make

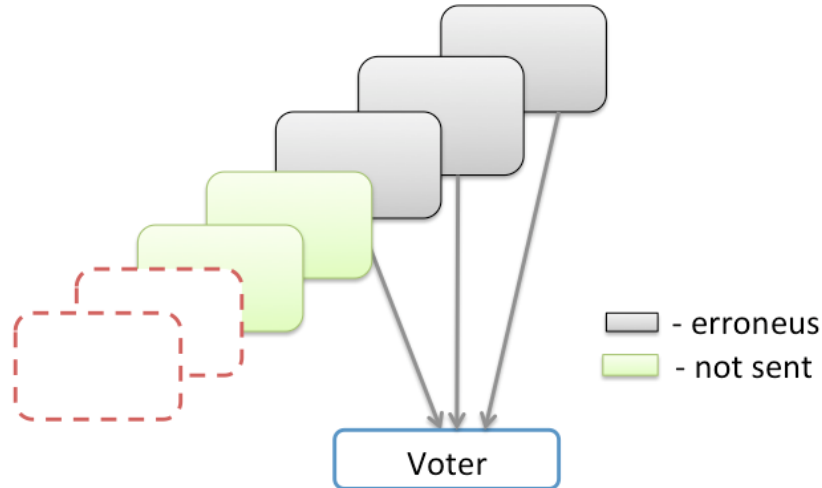


Figure 4.3: Voter Strategies:  $N$  is 7 and  $n$  is 5

a decision. If there is something preventing the siblings marked in green to send a message, then the voter would not make forward progress and be stuck.

We compare the performance characteristics of the two voting strategies in Section 7.4.

### 4.2.3 No Agreement

When a vote on input messages fails, an action needs to be performed based on the policy of the system. For instance, one of the actions may be to discard all the received ingress messages and restart faulty siblings. However, in a HA system, omitting egress messages is undesirable, so other policies may need to be enforced. Another strategy is to use incrementally more complex and computationally expensive Voting Implementations; a Failure of Majority Failure Comparison may lead to a Edit Distance Comparison or a Context-Aware Comparison. The system then needs to choose an egress message to send out. Since there no longer a "correct" message, it may be necessary to choose the "best" message that is the closest to all the other messages. In the case when there is no "best" message, a policy might be enforced to pick a random message and send it.

### 4.3 Routing Information Base

The two major Routing Protocols that we study are BGP[139] and OSPF[99]. Since OSPF is more accessible and is a widely used Interior Gateway Protocol, our research focuses towards it.

A Routing Information Base (RIB) is a routing table that exists in many different instances in the system. Usually, there is a single RIB, known as Global RIB, that resides in the kernel and from which a FIB (Forwarding Information Base) is generated. The FIB is used for lookup when a packet needs to be forwarded to a specific port. It is populated and managed by many different entities. Static routes can be inserted/updated/deleted by operators of the system. Routes may also change dynamically via Routing Protocols as they learn new information about the topology.

#### 4.3.1 Routing Table Structure

A routing table is generated from a Link-State Database (LSDB), which is a Graph Representation of a network topology. When the network is stable, all the nodes that participate in database exchange have identical LSDBs. When Dijkstra's Shortest Path First algorithm runs on the LSDB, a Routing Table for the OSPF protocol is generated.

Routing Tables are represented internally as Patricia Trees[99] with only two children for each parent. A level of the tree represents a bit position of a given IP address. An example of a Patricia Tree is shown in Figure 4.4, where a router stores the prefix **128.5.5/24** for **self**, **128.1/16** for **Router C**, **128.3.7/24** for **Router B**, **128.2/16** for **Router B**, **128.5.6/24** for **Router D**, and **128.2.4/24** for **Router A**. Let us assume that a packet needs to be forwarded to IP address **128.2.5.6**, and we need to determine the next egress router to forward the packet to. Usually, the most significant bit of the IP address is bit 0, and the least significant bit is 31. We start at bit 13, since at that bit we see the first discrepancy. Bit 13 is a Zero, so we go one level down and look at Bit 14, which is a One. Bit 15 is a Zero, therefore we keep going down the level until we hit a discrepancy, at bit 21. Since bit 21 is a 1, we go

all the way to the bottom of the tree, and try to match on prefix **128.2.4/24**. Since prefix **128.2.4/24** does not match with address **128.2.5.6**, we backtrack, looking for shorter prefixes, until we find a match. In our case, we find a match at Level 21 for prefix **128.2/16**, so the packet needs to be sent to **Router B**.

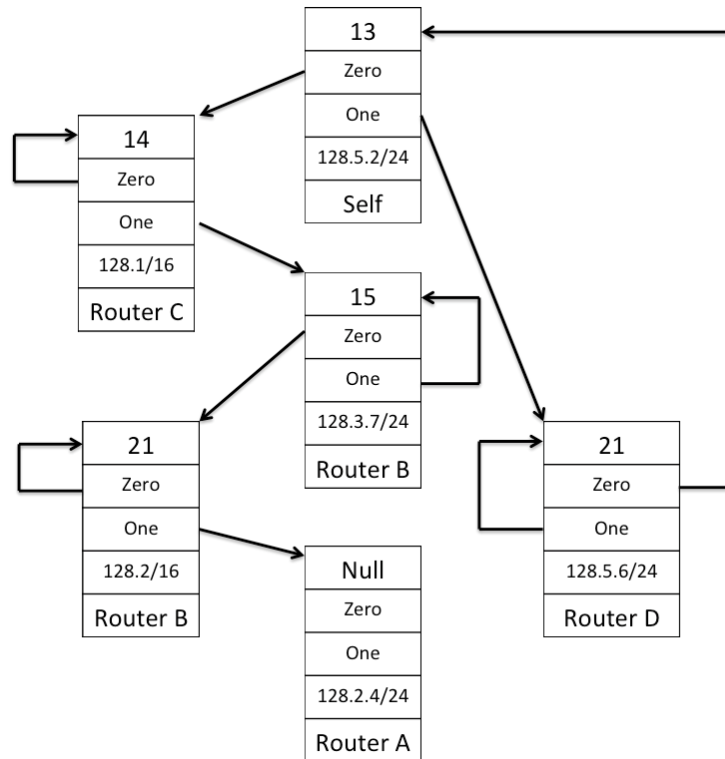


Figure 4.4: Sample Routing Table

### 4.3.2 N-Modular Redundant Model

Our process structure model is illustrated in Figure 4.5. An ingress message  $P_i$  enters the system. A splitter divides the input into multiple messages that are then received by each replica sibling. Each sibling operates on the message, which usually generates an egress message that needs to be sent to a subsequent system.

Usually, both a LSDB and RIB exist separately in each sibling. We define the local RIB as the RIB that exists in each sibling. When the local RIB changes, the change needs to be reflected in the global RIB as well. It is impossible for the siblings

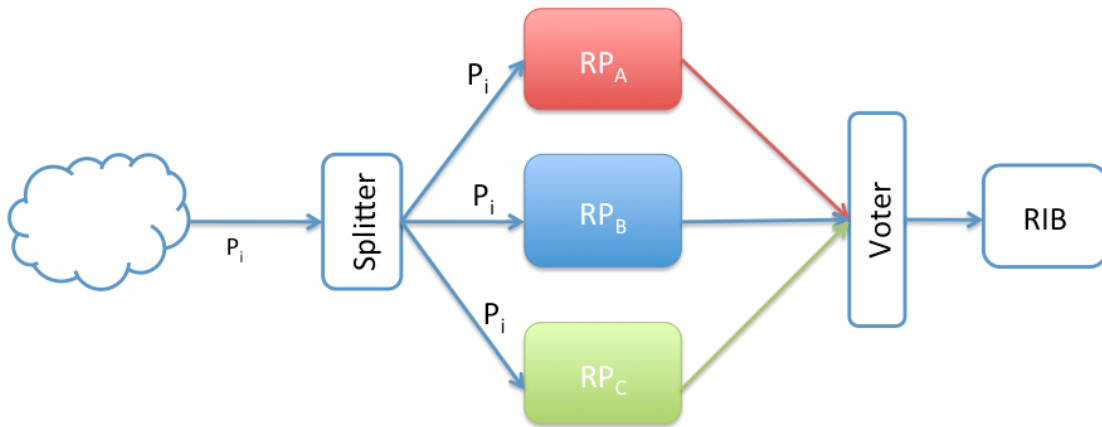


Figure 4.5: Process Structure Model

to send their whole RIBs on each change since it would overload the network. Thus they must have the same view of the RIB as that of the global RIB. When a change happens, they send only a message containing the change, which the global RIB operates on. An example of a message might have the fields (request, prefix, hops, metric), where the field request is either an add, delete or modify tag. The concept is illustrated in Figure 4.6

In a scenario where processes are faulty, it is possible for the siblings to diverge and output different messages, and this behavior needs to be detectable. Additionally, the process that updates the global RIB does not expect to receive a message multiple times. A voter, shown in Figure 4.5, can detect faulty inputs and collapse multiple inputs into a single one (see Section 4.2).

### 4.3.3 RIB Update Monitoring

A divergence in RIB change messages can be detected at the voter. As shown in Figure 4.6, the voter keeps track of the change in RIB Updates from each sibling with respect to every other sibling. The updates that diverge are stored in Delta Lists. We show the Delta Lists for each sibling with respect to  $RP_a$  in Figure 4.6.

When the size of Delta List reaches a threshold, we know that the two siblings have diverged. Each sibling stores a Delta List for all other siblings. If there is any

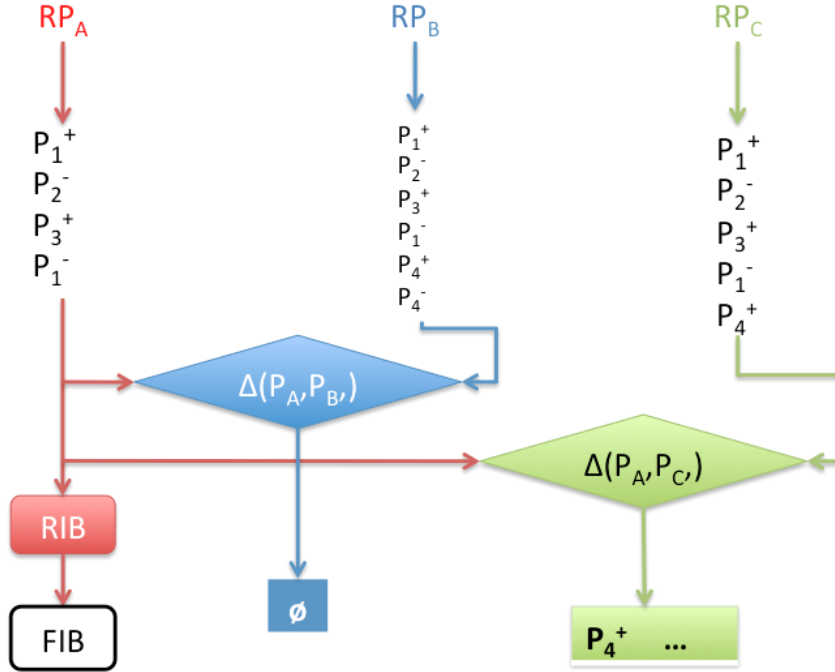


Figure 4.6: Change of Routes

sibling that has all its Delta Lists reach a threshold, we know that the sibling has diverged from all other siblings and is marked as faulty. In such a case, the voter needs to restart the faulty sibling.

#### 4.3.4 RIB Sibling Synchronization

The authors of FRTR[132] discuss the issue of designing a fast and bandwidth efficient mechanism to detect any inconsistencies between neighboring routers. The authors encode routing table data, which is periodically exchanged among neighbors. Similarly, core router software such as Cisco IOS XR have Route Consistency Checker[72][73] which compares the RIB against the FIB periodically. In contrast to these systems, we do not have a mechanism to directly compare and synchronize the RIBs in different siblings of a process group. We do this indirectly through monitoring of messages sent out. The justification is that too much periodic synchronization among siblings will slow down the responsiveness of the system.

## 4.4 HA Mechanisms

A service capable of Non-Stop Operation needs to be able to persist its state among multiple failures, have visibility into the currently executing environment, and have some form of global state synchronization. These concepts are covered in the following Section.

### 4.4.1 Checkpoint/Recovery

Resiliency of HA Services is a critical trait of its Non-Stop operation. Recovery from process-level failures is achievable with checkpointing. Checkpointing occurs when an application periodically sends its state to reliable storage. When a failure occurs and the process needs to be restarted, it can recover its state from previous checkpoints.

Checkpointing can occur either at

- User level - if transparent checkpointing is unnecessary, the application programmer can define the state that he wants to checkpoint, and it is application-specific. Otherwise, it is possible to achieve transparent checkpointing[7] by intercepting system calls, within some limitations.
- Kernel level - the operating system will need to be modified and the kernel restarted. The permissions required to perform these steps may be a restriction to some users. However, at the kernel-level it is possible for full transparent restoration of previous process state, even for restoration of process id and session id of a job[61].

The format of data being checkpointed can be

- Message based - only the ingress messages are checkpointed. When recovery occurs, the restarted process must replay its state from its checkpointed messages. While this approach is easier to implement and more flexible (for instance, the state may be replayed until a certain point only), the amount of checkpoint data grows quickly.
- State based - the process state of the replica is serialized and checkpointed[54]. When new process state needs to be checkpointed, the old checkpoint data may be overwritten or kept. While having less total data to store, the size of the checkpoint message may be too large depending on the program state. The serialized data is not compatible across programming languages and systems. However, the recovery procedure is simpler, since the current state can be loaded into the process right away.

- Hybrid - Message based checkpointing can be combined with state based checkpointing, where after a certain amount of checkpointed messages, the messages are collapsed into a process state, which replaces the previous messages. The collapse may be performed by a different entity so that the checkpointing entity does not suffer any performance penalty. A restarted process then applies any new messages to the left-over state. A similar technique used in OFBGP[48] described in Section 3.5 uses both periodic state snapshots and checkpointed ingress messages to recover its state.

Various ways exist of storing checkpoint data on reliable storage. Simplest way is to store the data on the local machine. However, the method prevents migration of processes when the system load is beyond capacity. If the machine hardware fails, the process as well as its checkpoint data is lost forever. A second approach is to checkpoint the data remotely to a server or NFS share[36], or to checkpoint it to a parallel file system[98]. The architecture introduces a single point of failure in the system, since the client needs to communicate with a remote resource over the network. If the remote resource or the network fails in some arbitrary manner, the checkpoint data becomes inaccessible. In terms of performance, the communications channel with the remote resource is a bottleneck, since multiple processes need to share the same channel when checkpointing data.

Another approach taken by authors in [46] is to checkpoint the data into a DHT (see Section 3.3.2), so that the data is shared among multiple nodes and thus is able to survive single-point failure and has faster turnaround time. This is the approach taken by us as well. When writing to a DHT, a client usually interacts with the replica of the DHT that resides on the same node, thus latency being equivalent to storing data on local machine. However, since the data is redundantly distributed, it survives a machine hardware failure. Figure 4.7 shows an experiment where we compared the latency characteristics of checkpointing to a DHT and Server. We wrote a Fibonacci Sequence Generating Algorithm, running in N-Modular Redundancy, and examined the latencies of both scenarios. As shown from the data, the maximum latency to checkpoint to a Remote Server is 32 ms and the maximum latency to checkpoint to a DHT is 22 ms.

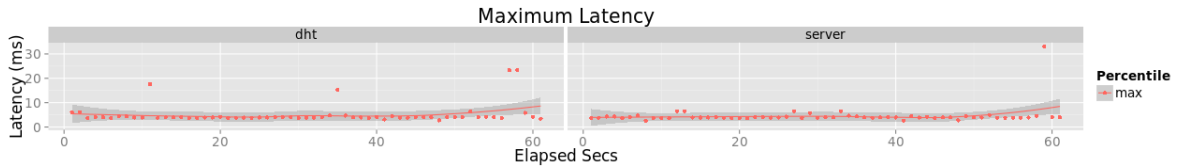


Figure 4.7: DHT and Centralized Server

#### 4.4.2 Resource Registry

In order for a process to be able to auto-discover other processes in the system, a distributed global process table must be present and accessible by all currently executing processes. The table entries can be grouped by common characteristics which their corresponding processes implement. A different grouping might be their creation time or their location (rack/line card). We call such a table a Resource Registry. When a process dies, it needs to be removed from the Resource Registry. This can be done either explicitly, due to a normal exit or implicitly, due to a crash. In the case of a crash, it needs to be detected by the resource registry, thus functioning as a failure detector.

Multiple implementations of resource registries exist. A process registry based on IPv6 addresses and Link-State Routing, is used in our implementation (see Section 5.6). Riak-PG[97] is a different example of resource registry, where the registry is based off a DHT. Gossip-like protocols[40][63] provide a weakly-consistent process membership, while Zookeeper[70] and etcd[111] can be adapted and be used as a strongly consistent resource registry. For greater analysis and comparison of weakly-consistent resource registries, please refer to Sections 5.6 and 7.1.

#### 4.4.3 Synchronization

When a process group is running, a synchronization mechanism needs to exist between siblings, so that they are not out of step with respect to each other. A mechanism for synchronization may be necessary for two processes that are not in the same process group, and need to share some common data.

## 4.5 Distributed Key-Value Stores

Distributed key-value stores are a useful and fundamental abstraction of a dictionary data structure. Since the dictionary is distributed, the data may be replicated at multiple locations and is thus resilient to single server failures within a data center. It is fundamentally different from a RDMS, where the data is unstructured and can be committed without having to preallocate a schema for it. Additionally, key-value stores target the design space of "always write" data stores[42], and are expected to be "write-heavy" as opposed to RDMS systems, which are "read-heavy". A Distributed Key-Value Store is sometimes also called Distributed Hash Table (DHT)<sup>2</sup>. For more information about DHTs please refer to 3.3.2 .

By being Highly Available and always willing to accept writes, even in the presence of network and node failure, the key-value store can become inconsistent between replicas, until the failure is fixed. The trade-off between availability and consistency is a well-known issue outlined in Section 2.8. For most key-value stores, it is possible to tune configuration parameters in order to achieve the desired balance, on a global, per-key, or even per-request granularity.

Key-value stores fit into Data Center architectures by adding capacity by horizontal scaling, both in speed and availability. Similarly to most data center services, adding cheap commodity PCs to the existing components in the data center allows more nodes to run replicas of the dictionary data, which increases the availability of the data. By having more nodes, the values for two different keys can be separated on different servers, allowing for concurrent reads/writes and speeding up the response time.

A relationship exists between distributed data store features and HA mechanisms:

---

<sup>2</sup> In the literature, a DHT is sometimes referred to as a system in which multiple hops are needed to reach the data (i.e. Chord), whereas a distributed key-value store is essentially a zero-hop DHT. Other times, no distinction is made between the two. In this dissertation, I use these terms interchangeably.

- Checkpoint/Recovery - Key-value stores serve as a good underlying framework for checkpoint/recovery. Checkpointing occurs far more than recovery, and thus write-time dominates read-time, a characteristic key-value stores are designed for. The state data is often unstructured and cannot fit well into schemas defined ahead of time. The data is easily indexable by keys; when message based checkpointing is used, the keys correspond to IDs of the messages; when state based checkpointing is used, the keys correspond to a certain milestone in the process state. It is appropriate to use an eventually consistent data store such as Riak[126] or Cassandra[86], especially in regard to Message Based Checkpointing, since the period between when the data is checkpointed and when it is read back during recovery is fairly large. We believe that using a distributed key-value store as a substrate for checkpoint data storage, a process is able to both minimize latency and maximize resiliency of the application.
- Resource Registry - A key-value store can be used to implement a resource registry. Riak PG[97] uses an eventually-consistent key-value store while etcd[111] uses a strongly-consistent key-value store.
- Synchronization - An engineering solution needs to be made whether the synchronization should be strongly consistent or eventually consistent. Strongly consistent synchronization refers to a solution where if a write is successful, all subsequent reads that are successful will have the updated value[131]. However, when unavailability due to node failure or network partition occurs, all writes and reads will fail until the system is healed. Eventually consistent synchronization, on the other hand, happens when a successful write is not immediately seen by successful read, which means that the read may be seeing stale data. Providing Read-Your-Writes Consistency[116] will ensure that the client is at least aware of read data that is stale, and has to decide whether to discard or merge data. Using both strongly consistent and eventually consistent data stores is necessary in order to achieve best performance during state synchronization. Strongly consistent solutions should be used when an immediately consistent global view of state data is required, and the number of replicas is not large. Eventually consistent solutions will ensure that the data is available, even during component failures.

Using a distributed key-value store as an underlying framework and abstraction mechanism, we can create HA services used by applications.

## Chapter 5

### GENERAL ARCHITECTURAL FRAMEWORK

This Chapter describes an environment that models a Distributed Routing Platform is described in this Chapter. We propose a platform made up of open source applications, which acts as a surrogate for a Distributed Router[8]. We operate a Resource Registry in the custom environment, which provides visibility into currently executing processes. The environment supports Process Migration, through state durability capable of withstanding single node failures. Finally, we conclude with our proposed system, HARRY, that runs in the environment.

#### 5.1 System Model

We model our design after a distributed clustering environment. As mentioned in Section 1, modern routing architectures are distributed computing platforms built with custom hardware. The platform consists of one or more chassis, each composed of multiple line cards that are interconnected and communicate via high-speed fabric (See Figure 5.1). The data plane is responsible for associating and sending incoming packets to external interfaces. The logic and forwarding decisions are usually implemented in hardware on each line card since the packets need to be forwarded as fast as possible. The majority of incoming packets are processed in the data plane hardware. As shown in Figure 5.1, each line card in the data plane is connected to an external interface.

A CPU typically resides on the each line card on the data plane. Its canonical functions are to collect statistics and record logs. There is also a computer that is typically associated with the data plane. The computer handles control messages destined for itself and messages that the data plane does not know how to process in hardware. The functions of the router's control plane (CP) execute in these computers.

The computer resides on a specialized line card called Route Processor (RP) with no external interfaces (5.1). The RP is used to execute higher-level functions such as routing protocols (IGPs, EGPs, etc.), or management capabilities(SNMP). There is a secondary idle RP (+1) that can replace the primary RP (1) when it fails. The control plane is responsible for learning dynamically changing routes, pushing them into the data plane, router management, etc.

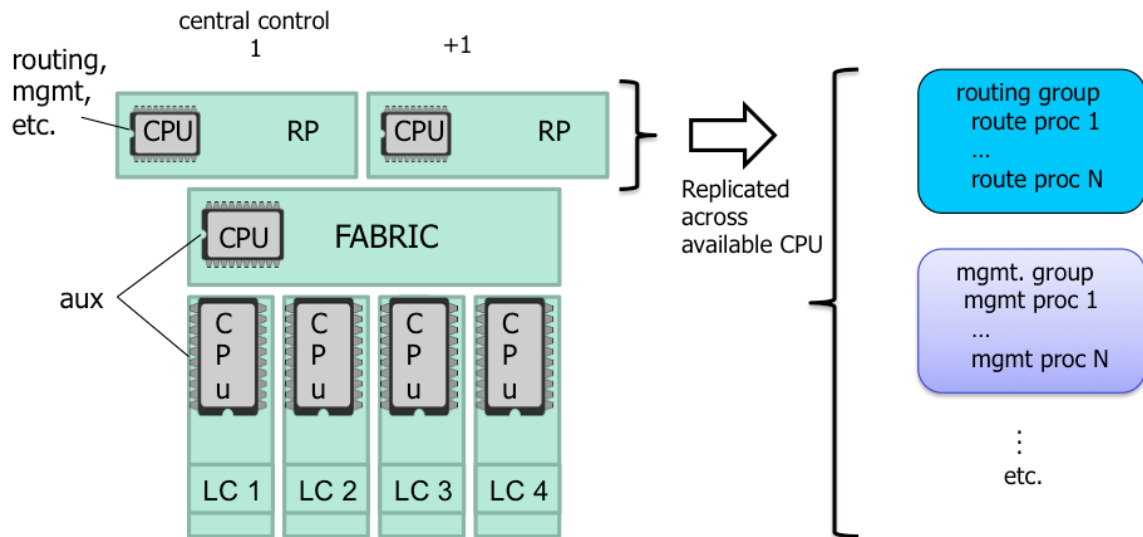


Figure 5.1: Core Router Line Card Model

A communications channel exists between the data plane (forwarding) and control plane (routing), called the punt path. This channel usually has low frequency of traffic compared to the switching fabric. The data plane connects to the control plane through a North-Bound Interface (NBI). Whenever a processor does not know how to process the packet in hardware, it punts it to the control plane.

In a typical configuration, the control processes execute in the central control RP line card. A Single Point of Failure (SPOF) exists in the system, so that if the RP fails, the distributed router would not have Control Plane functionality. While a Active-Standby (1+1) configuration helps alleviate the issue, real-time requirements by the industry in HA environment cannot be satisfied using these methods (see 4.1 and 2.3.2).

We attempt to solve the Single Point Of Failure problem by running the control process functionality in N-Modular Redundancy configuration. Rather than execute in a single RP, the CP processes execute on all available line cards, at the same time. For instance, a routing protocol runs as a group of N routing processes (see 5.1).

## 5.2 Proposed Platform

In order to verify our ideas, we need a hardware platform on which to test our implementations. Since we do not have access to the custom hardware of commercial vendors, we implement a surrogate of a modern routing architecture using Commercial Off-The Shelf (COTS) technologies. We prototype our implementation using conventional x86 hardware. We recreate the data plane, control plane, and the punt path using Linux systems. We impose a separation between the data plane and control plane, and connect the two planes together with the punt path.

We use the linux kernel as a vehicle for the data path, and therefore exercise its default code paths. Accordingly, our architecture is based around the system call conventions imposed by the kernel. Both at the data plane and the control plane, we are using code modeled after Quagga Software Suite[78], an open source network routing software suite.

Our architectural structure design is shown in Figure 5.2. The HA routing system consists of a chassis with 4 line cards, each having a CP and DP. Each line card is a pair of linux machines. Each world interface of the Router is the actual interface of the DP. Each CP/DP pair has a "Punt Path" connection between them. The architecture is modeled as a distributed computing platform, similar in many regards to a computing cluster in which state information is shared among nodes.

Our system is structured so that a DP of line card  $i$  cannot directly communicate with the CP of line card  $j$ . The DP and CP use link-local IPv6 addresses for communication. Thus the design is less complex and more straightforward. If the "Punt Path" of a certain line card is down, it can still communicate with the CP, but has to use other DP line cards to act as proxies.

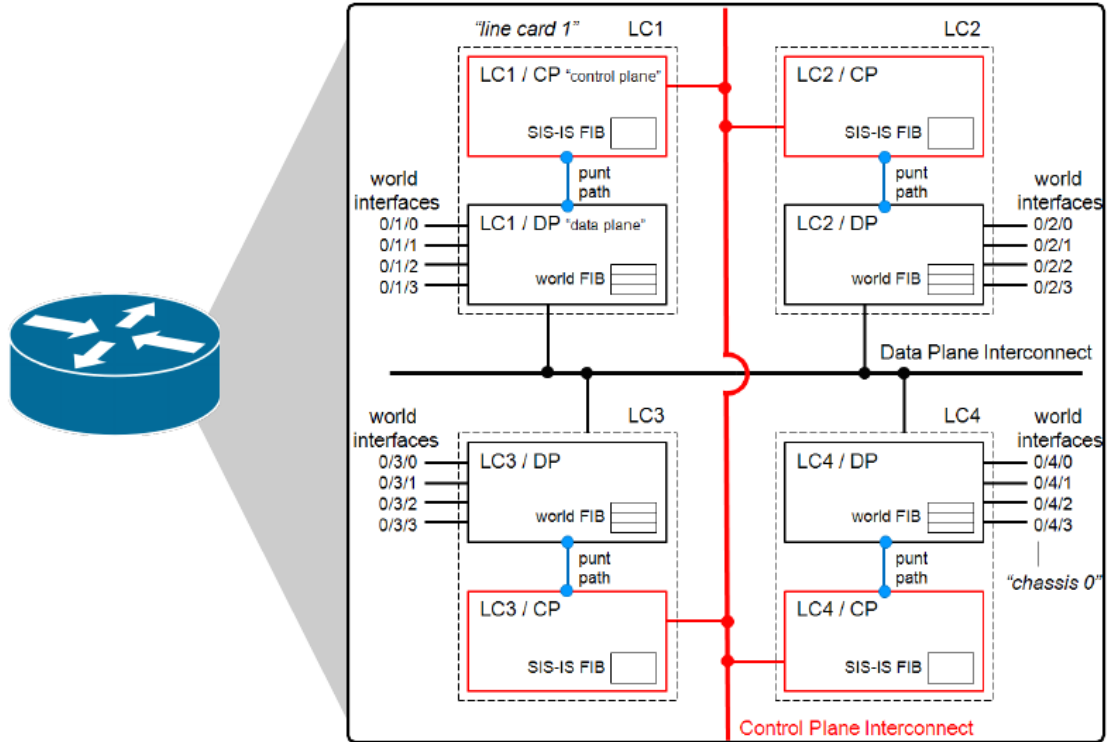


Figure 5.2: Routing Architecture Design

### 5.2.1 Comparison to SDNs

The clustering environment described in Section 5 and our proposed platform is similar in many regards to a SDN Network Architecture, described in Section 3.4. The forwarding is brought as close as possible to the interfaces, where it is forwarded the fastest. The punt path resembles the communications path between the Control Plane and Data Plane in SDNs.

Our implementation of the Communications Protocol, which we call RouteFlow, is based on OpenFlow[53]. The implementation details of RouteFlow are described in Appendix A.1.

Our work looks at how to make a single component, such as a Router Appliance, more resilient to failures. The goals that SDNs are trying to achieve are generally more encompassing in scope, where a network of appliances is the main focus. We believe that by focusing on only a single component, this approach is more susceptible to

deployment and less disruptive on the overall network. Additionally, we believe that some of components described in Section 6 can be applied to SDN architectures.

### 5.3 Control Plane

As mentioned in Section 5.1, processes are made to be n-modular redundant only at the control plane. We use the SISIS IPC for communication and process visibility. For more information on SISIS, please refer to 5.6.

There is a routing process group, of which a member sibling could reside on any line card in the CP, and it is the only process group that has a complete view of all ingress and egress ports. However, the group has no knowledge of the forwarding tables at each line card directly; the sibling group has an abstract view of the network. When the CP sibling needs to read or write a FIB (Forwarding Information Base) entry, it is altered by a CP controller so that internal routes are hidden.

### 5.4 Data Plane

Each line card DP has a unique forwarding table that is different from the forwarding tables of the other Line Card tables in the DP. For instance, in Figure 5.2, the LC1 DP FIB has different entry for port 0/2/0 from LC2 DP. At the DP, a process listens for incoming packets to determine whether to forward them to the DP components, or to "punt" them up to the CP.

We use the linux kernel for forwarding packets in the data plane. A well-known weakness of the system is that forwarding in software without custom built hardware is significantly slower than the capability of the Network Interface Card[114]. Since transit packets are the majority of packets processed in the data plane, this characteristic can significantly slow down the overall forwarding. In the following two sections we explore alternative data plane forwarding mechanisms, which can be substituted for forwarding via the linux kernel.

### 5.4.1 Route Bricks

The goal of RouteBricks[45] is to build open programmable router platform capable of high performance using conventional software routers. The authors' architecture is similar to ours, in the sense that each server is represented as a line card, and the servers interconnect among themselves in a mesh topology for internal switching of packets using load balancing techniques. Additional goal of the work is to increase parallelism in a single server by locking a particular CPU core to a queue in a multi-queue NIC. Our attempts at duplicating a system similar to the one outlined in the paper were not successful. The system depends on a certain version of Click Modular Router[82], and a specific type of Intel 10GE fiber-optic NIC. Click itself can be run in user-space or in kernel mode. User-space is not useful since forwarding will be performed at the speed of a software router, therefore kernel mode is ideal. However, user-space Click is more stable and reliable. The key feature enabling a potential for RouteBricks to run as a forwarding engine is the dynamic nature of Click elements. For example, the kernel-level mode of Click builds a directory of files, where standard read and write commands enable interaction with the underlying module. So it is possible to examine the contents of the routing table by just typing "**cat /click/rt/table**" in a command-line shell. Similarly, to append a route to the Click Routing Table a user can execute "**echo '128.150.1.1/24 1' > /click/rt/add**". Another interesting feature is that all of the routing table state in Click do not leak out into the kernel, so unlike Quagga an entry in the Click Routing Table is not seen in the linux kernel routing table. This enables someone to build a "Click-aware" implementation of a program, which can run alongside a regular application without fear of corrupting the routing table, which can be useful for prototyping.

### 5.4.2 NetSlice

Similar to RouteBricks, NetSlice[94] claims that it is able to parallelize packet processors linearly with respect to the number of cores the system has. It also supports our observations from experimentation with RouteBricks that user-space packet

forwarding does not enable line speed forwarding. NetSlice exposes an API similar to raw sockets. The functionality is enabled by loading a kernel module. The API allows a user task to choose which CPU core it will run on. A user may decide whether to bypass TCP or IP de-encapsulation. This configuration may be used for performing deep-packet inspection, as illustrated in [94]. Through debugging and experimentation, we were able to successfully operate a server with NetSlice extensions, where we were able to forward UDP traffic at line card speeds. In our opinion, NetSlice is more mature and stable than RouteBricks, and therefore is recommended for fast packet forwarding at line speeds.

## 5.5 Separation Between Control Plane and Data Plane

The Control Plane and the Data Plane are orthogonal with respect to the communications mechanisms. SISIS addresses, as described in Section 5.6 are not exposed to any process in the Data Plane, and the Data Plane does not know the type of addressing mechanism the Control Plane uses. The design of the system in this way is a necessary requirement. The SISIS address space cannot leak out and be publicly accessible by any external agents, since then it would be easier for them to exploit the system. The data plane should not have visibility into the addressing scheme used by the Control Plane, so that it is not ambiguous whether certain address belongs to an external agent or the control plane. The benefit is that external addresses and SISIS addresses do not have to be disjoint, a policy that is impossible to implement in reality. The Control Plane could easily switch to another form of communication mechanism without changing anything.

## 5.6 SIS-IS

A resource registry (see Section 4.4.2) is realized through SIS-IS (System IS-IS), a routing based resource registry<sup>1</sup>. Any process that wishes to be listed in the registry is assigned a unique IPv6 address based on the process type and some unique host

---

<sup>1</sup> The basis of this work is outlined in [118] and analyzed further in [9]

(machine) information. Existing link-state-routing protocols then redistribute these IP addresses, so that all other hosts running the routing protocol will have an identical list of all registered processes, represented by their IP addresses. SIS-IS is a framework on top of Quagga[78], and uses Zebra, a subcomponent of Quagga.

Key to our design of SIS-IS is mapping 128-bit IPv6 addresses to currently executing processes, so that each process has a corresponding address in its RIB. The address is distributed via the link-state routing protocol and is added to the RIB of all other nodes. By assigning well-known numbers to different characteristics embedded in the IPv6 address, which we call an SIS-IS address, each node can use its own RIB to find all processes of a given characteristic across the entire system. The characteristic may be a flag signifying which group the process belongs to, or a timestamp.

An SIS-IS address is only internal and cannot be leaked to any systems external of itself. Please see Section 5.5 for more information.

Figure 5.3 details the architecture of the SIS-IS Process Registry for one single node in a distributed system. In our system, the node is a single Control Plane Line Card, where the other nodes are the other Control Plane portions of the Line Cards. High-level processes, such as Process 1, 2 and 3, use the SIS-IS framework and may belong to an HA service, or may be auxiliary processes such as system monitors. Process 1 registers itself with the SIS-IS framework by communicating with its API. A SIS-IS Daemon creates an SIS-IS address on behalf of the process, and it is installed in the Kernel RIB. When Process 2 wants to query SIS-IS for currently executing processes, the API is used to retrieve the RIB on behalf of the process. Process 3 can register a callback function with the API, so that if the registry changes, it is notified.

The SIS-IS address, as shown in Figure 5.4, has a predefined prefix to distinguish it from any other IPv6 addresses in the local environment. A process type field specifies to which the group to which the process belongs to. Note that our system uses N-Modular Redundancy and therefore a process is usually a sibling that may have other currently executing siblings. The field is used by a process when it needs to determine how many siblings are currently alive, and for IPC such as leader election. System Id

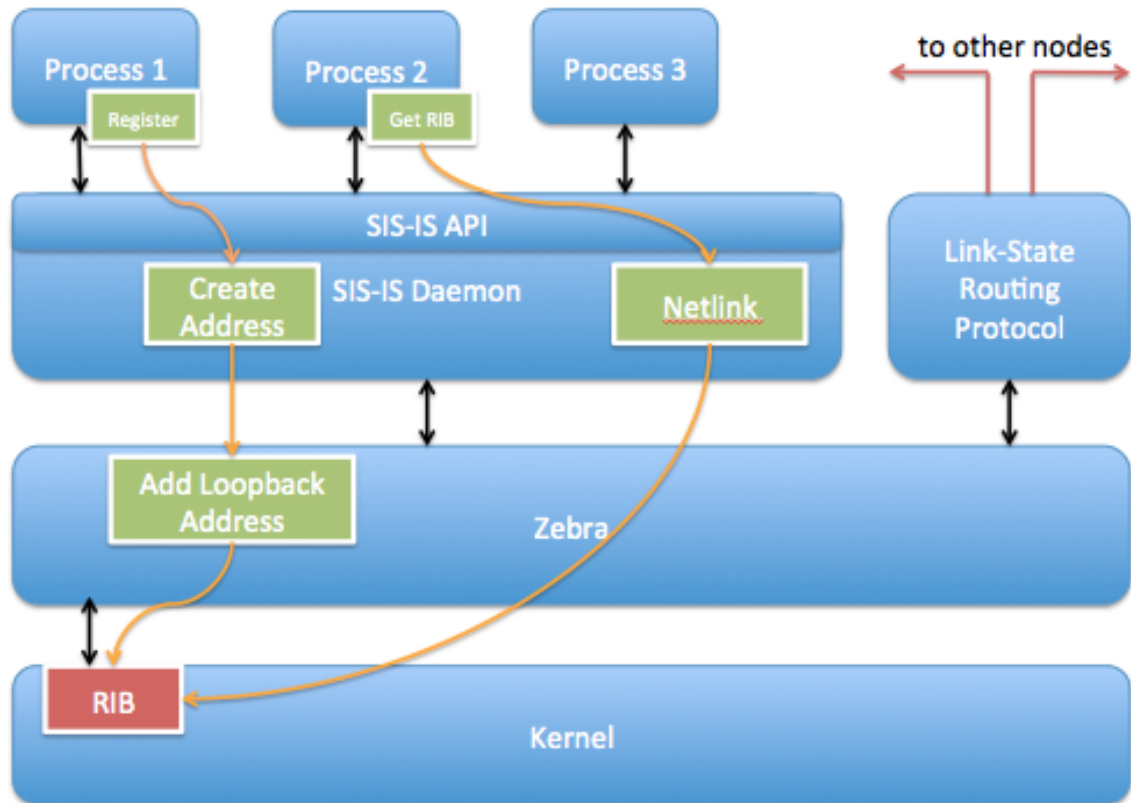


Figure 5.3: SISIS Architecture

field identifies the machine on which the current process is running. If we assume that Line Cards are numbered as shown in Figure 5.2, Line Card 1 has System Id 1, Line Card 2 has System id 2, and so on. The System Id field is used for process migration, and finding processes specific to certain Line Cards. The Controllers, for instance, are setup in such a way that there is one per line card. When the siblings start up, they need to know which Line Card (and external interfaces) a Controller corresponds to. Process Id field is the PID of the actual process, and is used to kill it if a monitor decides that it misbehaves. The timestamp field records process creation time. It is used in Leader Election to identify the longest running process. The concatenation of all fields, as shown in Figure 5.4, is the actual address used for IPC between all high-level processes.

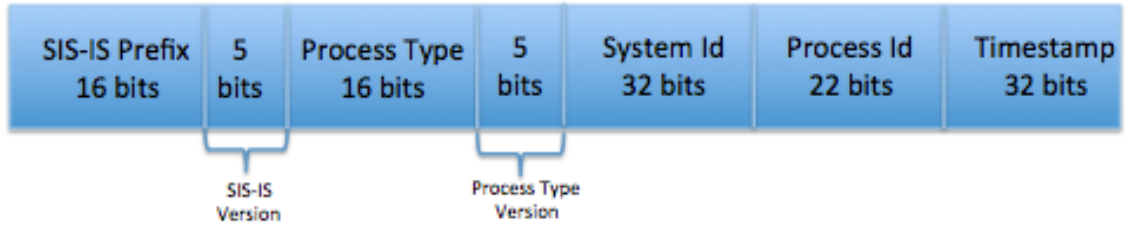


Figure 5.4: SISIS Address Format

When a process stops execution, either due to voluntary explicit exit or unexpected failure, the SIS-IS address needs to be removed from the registry. During a voluntary exit, the process can deregister itself before exiting. When an unexpected failure occurs, the SIS-IS framework deregisters the process if it has not heard from it in 30 seconds. When a timeout happens, the entry corresponding to the address is removed, and the change is propagated via the link-state protocol. When the whole line card fails, its RIB, including its state, disappears. The adjacent CP nodes detect the termination and remove all entries that originated at the failed host.

## 5.7 Different Fate Sharing Domains

Since N-Modular Redundancy is used, special attention needs to be taken in order to not have SPOFs. An SPOF may occur even with N-Modular Redundant systems, when instances run in a single Fault Containment Region. A Fault Containment Region[85] (FCR), or Fate Sharing Domain, is a region of a system that is neither vulnerable to external faults, nor causes faults outside of the FCR. If a fault occurs within this region, it cannot propagate to external failures, and failures of external systems cannot cause the system in question to malfunction. Thus, if several N-Modular sibling processes run independently of each other, but are executed by the same single CPU, the processes are run in a single FCR and thus all are vulnerable to SPOF by the CPU. As a general rule, in order to realize the benefits of N-Modular Redundancy, each sibling in a n-modular-redundant process group needs to run in a different and distinct FCR.

Since modern routing architectures resemble a distributed architecture(see Section 1), with multiple nodes and CPUs, it becomes possible in such an environment to run different processes in different fate sharing domains, where the failure of a certain component does not affect the rest of the system.

Similarly to a modern routing architecture, our proposed architecture allows for running each separate sibling in a different fate sharing domain. For example, in the control plane, a distinct sibling of a service process group can run in a different Line Card, and thus a fault within the Line Card will be contained, while the other Line Cards will still be operational. Thus the service still is operational through the liveness of the other process group siblings in the survived Line Cards.

In reality, not all faults can be predicted ahead of time, and thus it may be difficult to find the boundaries of a fault containment region apriori. Additionally, an individual can only control faults which he/she has jurisdiction over. A data center operator may have no control over a fault in a power plant that supplies the electricity for running the data center. In such situations, it is possible to define a level of granularity to the fault containment region, over which redundant processes can run. To predict

- A single line card failure, a sibling group may be run on multiple line cards of the system.
- A failure in the whole rack, bay or cabinet itself, or in the intercommunication fabric, a sibling group may be run on multiple racks, bay or cabinets of a data center.
- A failure, such as loss of power to a building, can be alleviated by running a sibling process group in distinct buildings.
- A natural disaster, such as landslide, or a regional ISP failure can be planned ahead of time by running distinct sibling process groups in different geographically located sites.

### 5.7.1 Process Migration

The system allows for process migration, where a sibling process can migrate to an alternate line card. Process migration can occur for various reasons. When a line card fails, all the processes executing on it die as well, and need to run somewhere else. For instance, in Figure 5.5, Line Card 5 (LC5) dies, and both the EGP and

management process siblings need to be migrated. Additionally, a process monitor can detect conditions which warrant migration of a live process, such as system overload when a given line card is oversaturated, or when two siblings share a Fate Sharing Domain. When a process monitor detects this condition, the process is killed, so that it may be restarted by a resource manager (see 6.2.3).

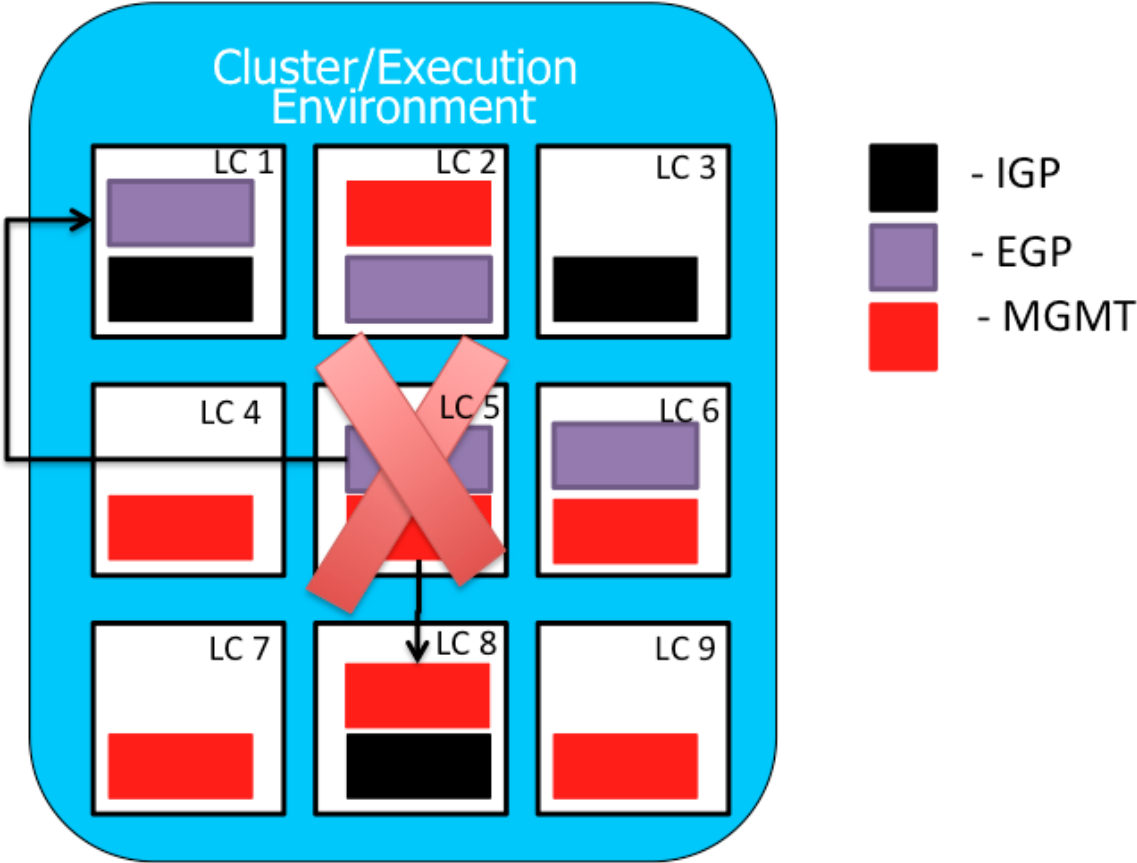


Figure 5.5: Process Migration

The resource manager is responsible for finding a good candidate on which to restart the process. A criteria the candidate line card must meet is that it must not be overloaded with too many currently executing processes, and it must be in a different fate sharing domain from its siblings. In Figure 5.5, the two successful candidates chosen are Line Card 1 (LC1) and Line Card 8 (LC 8).

## 5.8 HARRY

The research prototype, which we call HARRY (Highly Available Redundant Routing Yard)<sup>2</sup> is an implementation of a Resilient Routing Architecture. We have chosen to implement the architecture using emulation with deployable code on virtual machines to speed transition of the prototype to the field. We argue that our system can be used as a starting point to develop components in a Redundant Routing System.

Two types of packets enter HARRY: transit packets and control packets. Transit packets are meant to only traverse and be forwarded by the DP; for destinations other than the current router. The egress port for transit packets is determined using the local forwarding tables. In traditional routing architectures, all ports are on the same machine, thus at the egress port the transit packet leaves the system. However, in HARRY an egress port may be on a different machine, thus a machine that receives a transit packet on a DP needs to determine from its local FIB which DP line card to forward it to within the system.

Control packets are meant to be processed by the router. They are usually in the form of route updates, remote login connections, parameter updates, or statistics requests. One of our research focus is on making sure control packets are processed in a timely manner in the CP.

---

<sup>2</sup> Reference to a yard on which Highly Available Redundant Routing Components can be deployed and tested, similar to a shipyard.

## Chapter 6

### COMPONENTS AND CAPABILITIES

Our distributed routing system, HARRY, has multiple components that work together seamlessly. Their deployment in the execution environment is described in this Chapter. The capabilities of HARRY must meet demands of an HA system, as described in Section 4.1. These capabilities, such as Leader Election, Process Recovery, and Synchronization are covered in the following Chapter.

#### 6.1 Components

HARRY is composed of multiple components, working together to react to incoming messages and send out outgoing messages. The components are described below:

- The Punter - A necessary process that listens for control packets with a destination address of the system. The punter establishes the sockets required to receive the incoming data. For instance, if the system is capable of servicing OSPF traffic, the punter would set up a multicast listener for Layer 4 OSPF6 messages, or if it services BGP traffic, the punter would set up the appropriate TCP packet. The process resides on each DP Line Card. It is built to be modular, where various socket listeners could be added through a configuration file. The punter is the most lightweight process in the system of the other three described.
- Zebralite - Acts as the gatekeeper of the Data Plane. It sits at the bottom of the Punt Path and connects through a South-Bound Interface. A link local IPv6 address is used for communication with the other side of the punt path. In this way, the Punt Path connectivity is always on, without having to manually preallocate static or dynamic IP addresses. Zebralite has two functions:
  - Forward control packets from the punter process to the controller, and vice versa

- Retrieve necessary status information from the Data Plane and send it to the Control Plane. The FIB entries, interface status information, external address assignments, etc., are needed by Control Plane processes in order to make routing decisions. Zebralite is able to also modify these entries on behalf of the Control Plane.

Zebralite is modeled after Zebra, an abstraction layer in Quagga Software Suite[78] for the underlying Unix kernel.

- Controller - The Controller sits at the North-Bound Interface of the Punt Path. It takes packets from Zebralite and splits them to a Sibling Group through a splitter. The Siblings can be located via the SIS-IS API. At the Controller, a Voter is implemented. The voter takes in input from a Sibling Group, and makes a decision whether and how to forward the input to Zebralite. The voter is explained in more detail in Chapter 4.2.
- Groups of Siblings - The Sibling Group are the group of processes which are running in N-Modular Redundancy. A Sibling Group runs a Service, such as OSPF6, BGP, IS-IS, SNMP, etc,. The code is modeled after Routing Processes in Quagga Software Suite[78], but it is heavily modified. The Sibling Group has the most complex logic and state of all system components.

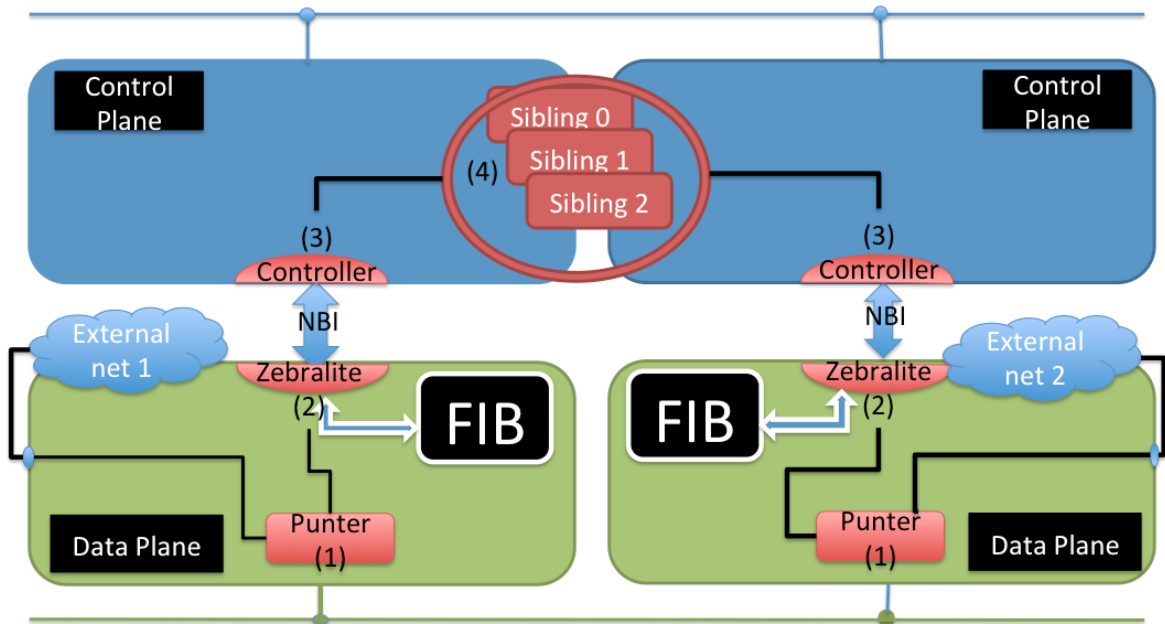


Figure 6.1: Different Parts of HARRY

### 6.1.1 A Typical Deployment

Figure 6.1 shows the operational components of the system. The Punter (1) listens for any control packets from external interfaces. Transit packets are forwarded via the kernel along the Data Plane. When the Control Data is extracted from the Packet, it is passed to the Zebralite (2) Process. Zebralite passes the control data to the Control Plane through the Controller's (3) North-Bound Interface. A Controller splits the incoming Stream of Data to a Multiple Sibling Group (4), and votes on the output of the siblings. The Sibling Group usually runs an implementation of a Routing Protocol. For each Control Plane/Data Plane Line Card Pair, a Controller resides on its NBI. Therefore, a System with multiple Line Cards has the Sibling Group communicate with multiple Controllers. Each Controller corresponds to the external interfaces that it influences. Even though controllers have affinity to the line card they control, any sibling of a process group may execute in any Control Plane Line Card. Diversifying all the sibling in different Line Cards is recommended, since they run in different Fault Containment Regions (see Section 5.7).

A TCP Connection is always persistent between the Controller (3) and Zebralite (2). When Zebralite initially starts up, it is initialized with interface address, state information, and Routing Table Entries. Zebralite retrieves the information by executing Netlink<sup>1</sup> commands to the kernel. When the Controller starts up, RouteFlow queries Zebralite for interface state and RIB entries (see Section A.2). A HA Sibling Group that needs the Data Plane State additionally sends information requests during its start up sequence to the controller. More information about HA Sibling Group start up sequence is detailed in Section 6.2.2 and A.3.

Please note that Zebralite does not distinguish internal from external interfaces. For instance, in Figure 6.2, the Data Plane Interconnect channel is not different in any way from the external interfaces. The Controller and the Sibling Group, on the other hand, need to distinguish between external and internal interfaces. The world

---

<sup>1</sup> A socket-based communications channel between the kernel and user-space applications in linux[102].

interfaces are known in advance to the siblings through configuration files. Therefore, routing protocols only run on world interfaces. Additionally, each Line Card is marked with a system ID, that its corresponding Controller embeds in its SIS-IS address. A configuration file specifies the external interface name and Line Card ID on which routing is turned on. When a forwarding path needs to be changed in the Data Plane global FIB, the Routing Protocol Service Group needs to send a distinct Route Update message to each Controller.

Figure 6.2 portrays a scenario where host N1 communicates with server N2. The default gateway of N1 is our system, HARRY, that is connected to two redundant paths, R2 and R3. The best and default path is through router R2. In the middle of communications, router R2 goes down and the path needs to be changed to use R3. The global FIB in all the Line Cards needs to be changed. In LC3, a network prefix corresponding to N2 needs to be inserted with default gateway as R3. In LC1, the next hop of N1's network prefix needs to be changed from the DP of LC2 to LC3. In LC2, the old network prefix needs to be removed since it is no longer valid. When a Routing Process discovers that its adjacent router has gone down and there is an alternative path, it sends a Route Withdraw to LC2 and LC1 followed by a Route Add message to LC3 and LC1. The Route Withdraw message that is sent to LC2 is sent unmodified. Similarly, the Route add message to LC3 is not modified by our system. However, the Withdraw Message sent to LC1 is altered and set with the nexthop of LC2, followed by a Add Message to LC1, where the nexthop is set to LC3.

The routing protocol itself is not aware that Route Add/Withdraw messages are modified. As far as the logic is concerned, all the world interfaces are abstracted away to look just like regular interfaces. A module which knows the Line Card numbering scheme intercepts the messages and modifies them. Initially, the design was to modify the messages in the controller, so that Sibling Processes are not even aware of which Line Cards an interface corresponds to. However, realistically this becomes unpractical, since then each controller needs to know the internal interfaces of all the other Line Cards. The amount of overhead in complexity of synchronization between controllers

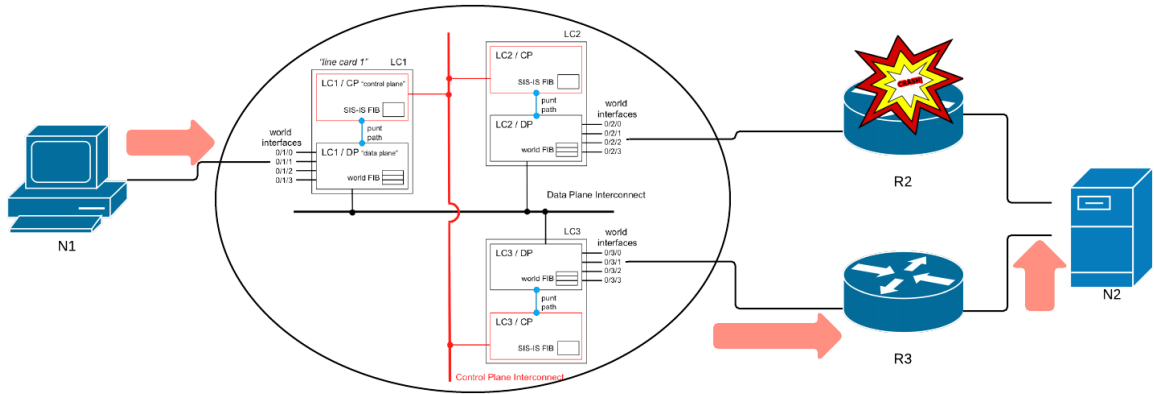


Figure 6.2: HA System in Operation

does not justify the benefit of decoupling the message transformation rules from the Siblings. Thus, the message transformation occurs at the Siblings. As opposed to a Controller, which only knows the interface info specific to its Line Card, the Siblings have knowledge of all interfaces, and therefore are able to transform messages easily.

## 6.2 Capabilities

The HA framework, HARRY, is developed in order to wrap non-redundant processes in a layer of redundancy and thus increase reliability to the entire system. The process state is managed with the help of a DHT (see 4.5) for stable store. This Section discusses in detail the capabilities of HARRY.

In Section 2.4, three different failure scenarios are presented. In subsection 6.2.1, we discuss how to detect the failure scenarios. Whether the process dies or is killed, it nonetheless needs to be restarted. Restarted siblings need to have their previous state replayed(see 6.2.3), where the state is retrieved from the previous checkpoints (see 6.2.4.1.1). The State Replay requires a Sibling Leader from whom restarted processes may retrieve messages that are missing. Additionally, we provide guarantees of consistency among the Process Sibling Group, explained in 6.2.4.

### 6.2.1 Failure Detection

Fail-Stop Failures, Byzantine Failures, and situations where forward progress is not made, can be detected by our framework. The novel methods are discussed below.

Fail-stop failures can be detected by SIS-IS callback. As mentioned in Section 5.6, any process may register a callback function that will be executed whenever an entry in the SIS-IS registry is removed. Thus, a monitor process can detect a Sibling Failure and schedule a Sibling Restart, outlined in Section 6.2.3. Initially, a leader process (see 6.2.2) was used as a monitor for any sibling failures. The leader would signal Torque?? to start a new sibling in Restart mode. Since moving to Marathon??, the feature at the leader was disabled and offloaded to Marathon, which can detect abnormally failed processes. It acts as a process monitor and sends an event notification to a subscriber when a process it has started dies.

Processes that are not making forward progress are detectable by a downstream process that is expecting the misbehaving process's output. A failure detector[33] is implemented at the Controller that can detect unresponsive processes. When a controller sends an outbound message to the sibling process group, a timer is started. A sibling process group is expected to provide responsiveness acknowledgement within 200 msec. The responsiveness acknowledgement is not an acknowledgement that the message is processed, but rather that the process is able to properly handle the inbound message and is not hung anywhere. Consider Figure 6.3. A controller sends a message to three siblings, Sib 0, Sib 1, and Sib 2; three timers are started: timer\_0, timer\_1 and timer\_2. Within 200 msec, the acknowledgements of Sib 0 and Sib 2 are received, and their corresponding timers are cancelled. However, Sib 1's acknowledgement is not received within 200 msec and its corresponding process is flagged as misbehaving. When the failure detector in the controller finds a misbehaving process, it terminates it so that the resource manager may restart it.

Byzantine Failures can be detected by a majority voter (see Section 4.2). We make the assumption that the data that is in majority is usually correct. The processes that are not in majority are assumed to have byzantine failures and are indicative of

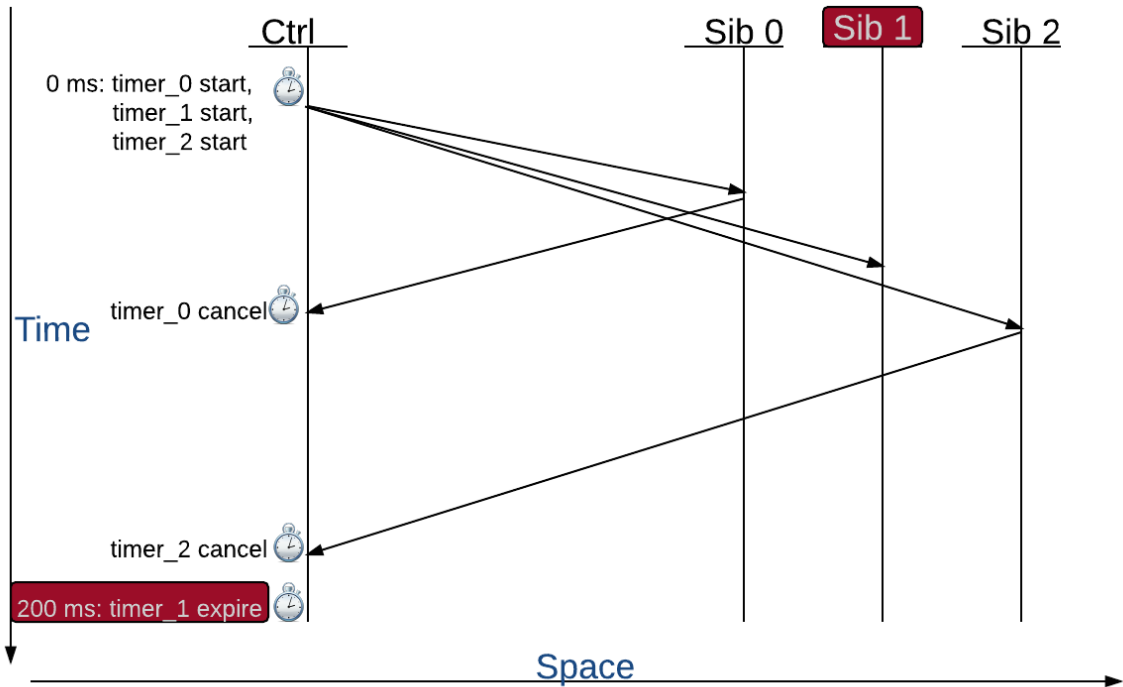


Figure 6.3: Process Timeout

abnormal behavior. Since the voter is at the Controller, the Controller makes the decision to kill a process that it flags as suffering from Byzantine Failure.

We do not consider the possibility of packets that are incorrectly received as a result of network errors, since we use TCP for all communications and assume that incorrect checksums alerts us about these situations.

### 6.2.2 Leader Election

Leader election is necessary for state replay (see 6.2.4.1.2) and sibling restarts (see 6.2.3). Upon process startup, the siblings are initialized with the Controller. For more information about Startup Sequence between a Sibling and Controllers please refer to A.3. Since an individual Sibling may have multiple Controllers that it communicates with, each Sibling waits on a Leader Elect message to receive from all Controllers. Once a Leader Elect Message from all Controllers is received, Leader Election begins. The siblings do not initiate Leader Election as soon as they start up, since

they do not all start up synchronized. The Controller ensures a synchronization point at which the Siblings are initialized with enough state data. The Controller only sends a Leader Elect message when all the Siblings have initialized with the Controller.

When Leader Election starts, each Sibling verifies with SIS-IS that the right amount of siblings are up. Then, each Sibling retrieves all the SIS-IS routes of its own sibling group and compares the Timestamp Field (for SIS-IS refer to 5.6). Each sibling has its own local view of the registry. From the local view, each Sibling sets itself to the leader, a Sibling that has the oldest timestamp.

It is possible that some siblings register with SIS-IS at the same time and receive the same timestamp. Each sibling has a unique replica ID associated with itself that it retrieves from a configuration file. If a timestamp conflict occurs, the replica ID can be used as a tiebreaker when the ID of the other sibling is known. If unknown, multiple leaders are elected by the algorithm. This is resolved by exchanging a Replica Exchange Message between all the siblings. A Replica Exchange Message contains an id and binary value indicating whether the id is a leader or not. The two outcomes of Replica Exchange is that the IDs of all the other siblings are learned and all the siblings have global view about the identity of the leader. IDs just learned from Replica Exchange are used for Leader tiebreaks. If a sibling A sees that it has marked itself as leader and another sibling B has marked itself leader, then A unsets itself as leader if B's ID is smaller.

A death of a leader results in the Siblings running leader-less. For this reason, when a failure leads to a sibling restart (see 6.2.3), Leader Election is always redone. Even if leader reelection does not change the current leader, it ensures that the restarted sibling has up-to-date information about the sibling IDs and who the leader is.

### 6.2.3 Sibling Restarts

When a sibling dies, either due to a fail-stop failure or as a result of being killed by monitor process, a mechanism is required that automatically restarts the dead process. Please note that State Replay, the next stage where processes replay

their state to a known good state, is discussed in Section [6.2.4.1.2](#) and is not discussed here.

We have developed two techniques for Sibling Restarts. The first and initial technique uses the leader. The leader periodically monitors the SIS-IS registry to verify that the quantity of redundant siblings currently running is equal to the amount required by the system. When the number drops, the leader spawns enough necessary processes. The unique ID preassigned earlier is reused for the restarted process. Only the leader is allowed to respawn siblings to prevent oscillations that may occur in the number of restarted processes. For instance, if both the Leader and the Controller is allowed to restart, multiple non-coordinating restarts would result in too many processes being respawned, which later would have to be killed. When a leader itself is killed, another round of leader election chooses a new currently running leader. The newly promoted leader is now responsible for restarting any processes that died.

Initially, Torque[\[121\]](#) distributed resource manager was used. The leader issues a Torque command to start a sibling in restart mode. When a process is restarted, it needs to perform state replay (see [6.2.4.1.2](#)) and thus needs to be started in a different mode than canonical process startup. A critical requirement was that we support Process Migration (see [4.1](#) and [5.7.1](#)). We could not get Torque to support Remote Process Respawn, where the Leader on one Line Card tries to spawn a Process on different Line Card, thus were unable to support Process Migration.

Due to the lack of documentation and tutorial, we abandoned Torque and switched to Marathon, a framework built on top of Mesos[\[67\]](#). Marathon does support Process Migration[\[3\]](#), and can act as a process monitor. Marathon and Mesos is well supported and maintained. When any process that Marathon has started dies, a message is sent to a script, which tells Marathon to automatically start the process in restart mode.

## 6.2.4 Consistency

Our system uses both strongly and weakly consistent coordination among its multiple processes. Our system tries to avoid the use of only strongly consistent synchronization among all replicas because of its high throughput requirement and impracticality in horizontal scaling-out. As suggested by other researchers[14], Coordination is expensive, and thus we attempt to use coordination-avoidance consistency whenever possible, only limiting strongly consistent coordination to situations where it is absolutely necessary.

In [15], authors suggest that the decision on when replicas should coordinate is imposed by application logic invariants. In order for some invariants to be satisfied, the underlying processes require coordination. Other invariants may not necessarily require coordination in order for them to be satisfied. The property, called I-confluence (invariance confluence), states that in order for a system to be I-confluent, all local writes to a replica must be globally invariant-preserving.

We use Riak[126] for a Distributed Key-Value Store (see 4.5) on which to store checkpoint data. An instance runs on each Control Plane Line Card, and each instance synchronizes using weak or strong consistency (see 6.2.4).

### 6.2.4.1 Weak Consistency at Siblings

Each sibling has its own bucket, or section, of the DHT (3.3.2) to which it checkpoints (6.4). A sibling can read or write to its own bucket; it can only read from another sibling's bucket.

#### 6.2.4.1.1 Checkpointing

Data is checkpointed at user level, therefore we specify at the application-level the data that needs to be checkpointed. The checkpoint data itself is message-based. Only the ingress messages are checkpointed. The key is the ingress id (see 6.2.4.2 for information about how ingress id is generated) and the value is the actual message contents in the Key-Value Store.

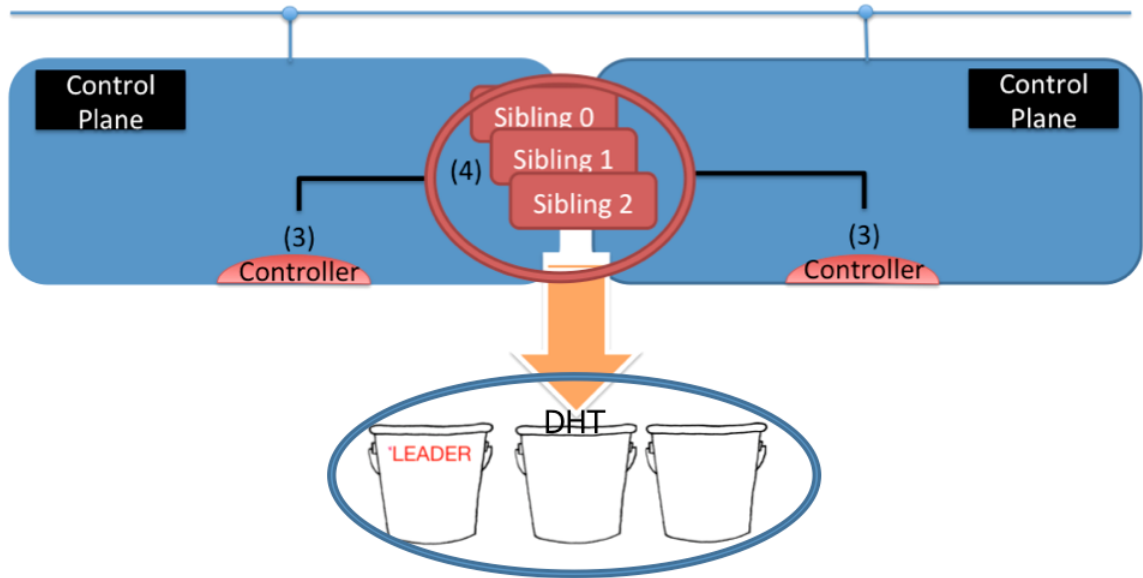


Figure 6.4: Checkpointing

Whenever a sibling process receives a message from the controller, it is processed, and then checkpointed (written) to the DHT in its own bucket. The sequence number or ingress id, which is used for key, is a monotonically increasing integer that provides a total ordering of all received messages. Between any two messages, if first message has a lower sequence number it means that it has been received earlier by the system. A flag is set in the bucket indicating whether its corresponding sibling is the leader or not. Since DHT is redundantly distributed on all line cards, the checkpoint messages are accessible on all of them. The Line Cards synchronize using eventual consistency (weakly consistent coordination). Because of the requirement on the system that a sibling replica only checkpoint data to its own bucket, our system is I-confluent in regards to its checkpointing capability. Since there is only a single source of writes to the same bucket, there is no risk that a merge operation may lose any write. The I-confluence property ensures that it is safe to use coordination-free, available eventually consistent system (the "AP" in CAP).

#### 6.2.4.1.2 State Replay

A sibling that is restarted initially starts up in Restart Mode with non-valid state. Its state must be caught up to all the other siblings in the process group. Its order of operations is:

1. Initialize itself with the controller. The process is described in more detail in [A.3](#).
2. Any incoming messages from the controller are enqueued. During this time the sibling does not send responsiveness acknowledgements back to the controller, and the controller needs to know that the sibling is in Restart Mode.
3. The sibling selects and processes all its previous messages from the DHT with its unique replica ID. The retrieved messages are the old messages that the previous process checkpointed before exiting. At this point, the sibling has caught up to where it last was before it died.
4. The sibling process determines how many messages it is missing, by looking at the sequence numbers from its last processed message to its first enqueued incoming message. All the messages with sequence numbers in between are missing. The missing messages are selected from the leader. At this point, the sibling has caught up its state to the point of being able to process its enqueued messages.
5. The enqueued messages are processed until the queue is empty.
6. The restarted process has caught up to the rest of its siblings and is ready to process incoming messages. The process is no longer in restart mode. The controller is notified that it is now in the Ready State.

In [\[15\]](#), the authors claim that any two operations, at least one of which is a write, will result in read/write conflict. Thus, during state replay, it is possible for a restarted process to not have a complete view of all the checkpointed messages by the leader. However, since we are retrieving only a subset of the leader's checkpointed messages, we assume these checkpoint messages are visible by the time state replay is initiated.

#### 6.2.4.1.3 State Catchup

The siblings do not synchronize among themselves before sending messages. While this helps us reach the Real-Time Response Requirements outlined in [4.1](#), we allow for the possibility that multiple siblings of a process group may not have synchronized state machines and that some siblings may fall behind. In order to try to avoid

frequent sibling restarts, which is expensive, we use the eventually consistent DHT for state machine synchronization among siblings.

During checkpointing, each sibling also puts into the DHT the current sequence number it is working on. From this information, any other sibling can ascertain whether it is falling behind or not, and if it is, execute a modified version of State Replay algorithm defined in 6.2.4.1.2, where Steps 1 and 3 are skipped. Please note that our system is very similar to systems described in [116], where synchronization happens in the background off the critical path, and occasional conflicts are fixed after they happen. It is easier to preserve the real-time response requirements in such systems.

**6.2.4.2 Strong Consistency at Controller - Monotonically Increasing ID**

The following section argues that Strong Consistency is necessary in few select cases, where it is unavoidable.

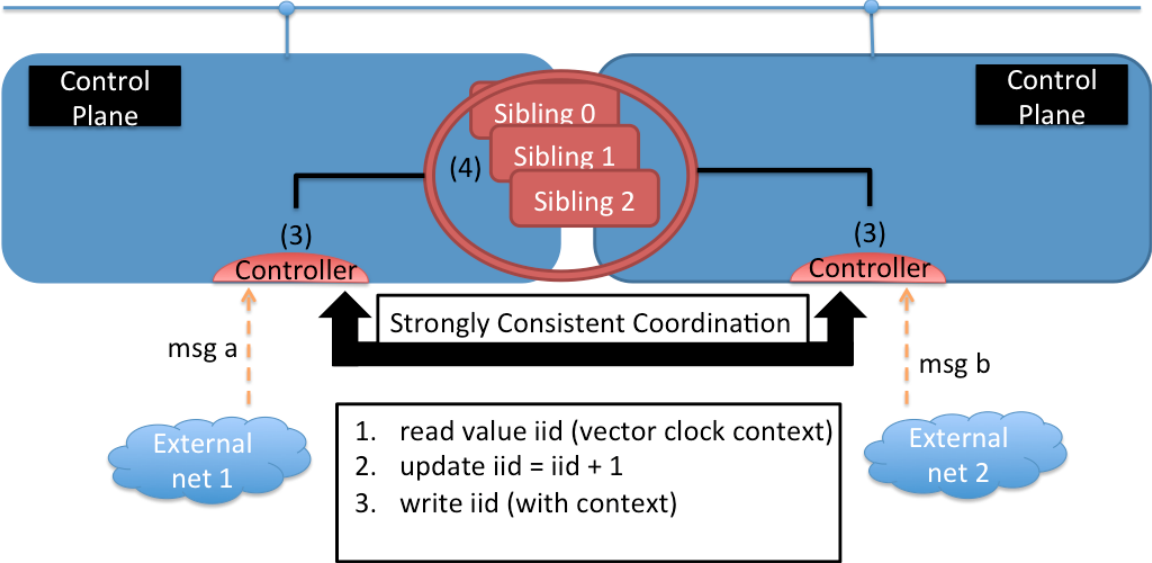


Figure 6.5: Strongly Consistent Coordination

For example, consider a database system that must allocate a unique ID to any new employees in a company. The database that ensures correctness needs to maintain an application-level invariant on behalf of the application, mainly that no two newly allocated unique IDs are the same. If the database is distributed in different locations

and weakly consistent coordination is employed, it is possible for the application-level invariant to not be preserved. Consider the situation where the unique ID is initially 0, and the database is redundantly replicated in two geographically distinct locations, NY and LA. A situation can arise where at roughly the same time, two new employees are created at the company, one at the NY office, and one at the LA office. Assuming monotonically increasing IDs, the employee ID at NY office is created before the employee ID at LA office, and its ID of 1 is set in the database. However, if weak, eventually consistent synchronization is used, the coordination update from NY to LA will occur only after the new updated ID of 1 is committed into the NY replica of the database (see 2.7.2). Before this update reaches LA, the weakly consistent database allows for the employer ID in LA to be set to 1, thus violating the invariant. The example is similar to an example proposed in 2.7.2, where writing to two queues from different processes is not reflected right away when reading from those same two queues. If some invariant requires that the two queues always have identical values, they must synchronize before that value is committed into the queue, not afterwards. To ensure that the application-level invariant is preserved, higher level of coordination is necessary. The two databases in NY and LA must coordinate on the next unique ID before it is committed to the database. The commit in this system needs to ensure that it is a global view commit.

Our system has a similar invariant that must be preserved for all ingress messages that enter the system and are destined for the Siblings. Since the messages are checkpointed, they need to have a unique monotonically increasing ingress ID (iid). The system will overwrite checkpoint messages if there are non-unique keys that map to different messages. However, the ingress messages have distinct sources of origin and thus cannot be simply tagged by some single arbitrary process when they enter the system. We decided to tag the ingress messages with a unique ID at the Controller, however since there are multiple Controllers, they need to synchronize.

Initially, Etcd[111] which has a working implementation of Raft, was used. After a few months, Riak 2.0[18] came out with support for Strong Consistency through

implementation of Paxos, which we switched to since it was easier to integrate with our system. Strong Consistency in Riak[17] is implemented through the use of Causal Context. A causal context is a string representation of a vector clock that is used internally to track causality. Before writing a value into a Strongly Consistent Bucket, the client needs to perform a read on the value in order to retrieve its Causal Context. Thus when writing the value back, the write operation also needs to supply the Causal Context along with the value it is trying to write. If there are two simultaneous Read/Write operations by two distinct agents, both Reads will return the same Context, however only one agent will succeed in writing. The agent that fails has to retry a Read/Write operation with different context.

The unique ID generating algorithm is portrayed in Figure 6.5. When an ingress message arrives at any controller, the controller locally performs the Strongly Consistent Coordination Algorithm. If the write fails, the algorithm is retried until it succeeds. When the write succeeds, the updated incremented iid can be used to tag the ingress message and send it along to the Siblings.

By synchronizing at the Controller rather than the Sibling, we avoid the potential overhead that may be incurred when Siblings synchronize using Strong Consistency. Please note that the number of Controllers is dictated by the amount of Control Plane Line Cards in the system, however the amount of Siblings is application-specific depending on how much redundancy is needed by the system. Thus the amount of Siblings can be much greater than amount of Controllers, potentially increasing the latency of the system. Even though coordination for each ingress message does slow the system down, Line Cards in a distributed system are usually co-located in the same Data Center, and thus the latencies are not arbitrarily large.

## Chapter 7

### EVALUATION

After designing and building the system, we needed a form of verification that ensures our system continues operation during the presence of faults. We begin by evaluating our IS-IS Based Resource Registry and contrasting it to other Resource Registry implementations. Then we look at responsiveness of HARRY to RIB Update Changes, and compare it with other Routing Frameworks. For the next test we introduce faults into our system and observe how long it takes for recovery to complete. We also find the smallest possible fault inter arrival time. Finally, we experiment with different implementations of the Voter, and provide some insight into their tradeoffs.

#### 7.1 Resource Registry

As discussed in Section 4.5, a Resource Registry can be implemented using a Distributed Key-Value Store. In our work, we compare SIS-IS (see Section 5.6) to Riak PG[97]. During our testing, we attempt to model the experiments and structure of the two systems as close to each other as possible.

Table 7.1: SIS-IS Registration Delays (ms)

Host	Min Registration	Max Registration	Average
1	0.26	0.54	0.30
2	0.95	4.23	2.03

Table 7.2: SIS-IS Deregistration Delays (ms)

Host	Min Deregistration	Max Deregistration	Average
1	0.26	0.38	0.29
2	1.22	6.62	2.27

We setup our experiment by creating a five node cluster, interconnected by Ethernet. We have three nodes running Quagga with a high-level application process such as OSPF and an SIS-IS Daemon. Additionally, we have all five nodes connected in a DHT cluster with Riak PG. We have N set to three, with both W and R set to one. Setting the data replication (N) to three in a cluster of at least 5 DHT nodes provides reasonable data safety even if two nodes fail[79].

The first behavior which we measure is the time to register and deregister processes. We simultaneously start up 30 processes and measure the time taken from initiating a process registration using the SIS-IS API, until the address actually shows up in the RIB. We then kill the previously started processes and measure their deregistration delays. The deregistration delay is the time from sending a deregistration message to the SIS-IS API until the entry is actually removed from the RIB. We observe the convergence times both at the local host and at a second node. The results are shown in Tables 7.1 and 7.2. We also start 30 Riak PG processes and time their responsiveness as they joined and left a group. We set N equal to 3, and varied the R and W values. Initially, R and W values were set at one, meaning only one node needs to respond to a read or write. We then change R and W to quorum, which in our case meant that two of three nodes need to respond. Finally, we set R and W to all, meaning that all three nodes need to respond before a read or write is deemed successful. Table 7.3 and 7.4 show the results.

Table 7.3: Riak Registration Delays (ms)

Conf	Min Registration	Max Registration	Average
one	0.70	1.05	0.75
quorum	0.70	8.79	1.21
all	0.66	6.47	0.92

Table 7.4: Riak Deregistration Delays (ms)

Conf	Min Deregistration	Max Deregistration	Average
one	0.86	1.26	1.12
quorum	0.75	4.30	1.13
all	0.95	9.56	1.32

The results confirm our hypothesis that when  $R$  and  $W$  is equal to one, the two systems act most similarly to each other. The SIS-IS system, on average, is faster than Riak system, both in registration and deregistration delays. However, when we vary  $R$  and  $W$  to quorum or all, we notice that the delay is not much greater than when it was set to one. That is because the requests and the responses are sent/received independent of each other. On the other hand, it took significantly longer for the SIS-IS addresses to propagate to the second node. It took about 2 ms for the addresses to show up in the second node's RIB. This is a byproduct of Zebra's architecture. We initially discovered that Zebra utilizes a queue for processing RIB requests, and uses a hold time before processing the queue. This value was decreased to 1 ms in Zebra to minimize delay. The observed 2 ms delay comes from the first node holding the RIB entry for 1 ms plus the second node holding the RIB entry it received from OSPF for 1 ms.

The second behavior measured is the delay in searching for a given process in the registry. The measurements are performed by timing the call of the SIS-IS and Riak APIs to retrieve a process list corresponding to certain criteria, i.e. the target process list. The processes that do not correspond to the given criteria are the extraneous processes and are members of the extraneous process list exclusively. We perform tests where we vary both the size of the target process list and the size of the extraneous process list. Initially we had no extraneous processes and varied the target process list from 1, to 5, then 25, and finally 100 processes. We performed this set of tests again, with the variation of adding 50, 100, 150, 200, and 250 extraneous processes respectively. For each test with a given configuration of size of target process list and extraneous process list, we ran each test 1000 times and averaged the results. The Riak DHT was run with  $R = W = 1$ , which are the least constrictive values for retrieval from a DHT. The results are shown in Figure 7.1 for SIS-IS registry and Figure 7.2 for the DHT registry.

Figure 7.1 indicates that the time to search in SIS-IS is affected by both the target process list and the extraneous list. Since the whole RIB is returned, and then

a linear search<sup>1</sup> is performed during which the fields in the SIS-IS address are matched against the given criteria items, the delay to retrieve and match entries from the RIB increases linearly with respect to the total number of processes in the system. On the other hand, in Figure 7.2, the entries in Riak are stored in a hash table with the primary key being the process group. Since a search is not performed over the whole process space, the search in Riak for target process list is not affected by extraneous processes. At approximately 1,300 total processes the simple linear version of SIS-IS will slow to that of the DHT. For these use cases, SIS-IS would be augmented with additional indexing to reduce search time. The process registry in SIS-IS is retrieved much faster than in Riak PG. This is in part due to ongoing flooding of state data to all nodes, so that often the SIS-IS API is able to retrieve the data locally.

Note also the DHT's ability to search the target process list is only for searches against the primary DHT key, e.g. the target process group. Any other search of characteristics will be slower as data must be retrieved from all the DHT nodes. The SIS-IS process registry is more powerful as it allows queries to be for more than just groups, in fact, any characteristic is possible as long as it is defined inside the address space. An SIS-IS process can also see that characteristic without having to consult a database, if the requested SIS-IS address is known. On the other hand, in an implementation where the addresses have no meaningful embedded information, each characteristic needs to be consulted to find whether the address is a member of this characteristic. An additional feature in SIS-IS is the availability of the link-state routing metric, a measure of logical distance, which allows searching for the closest process of a given characteristic set when the nodes are in a complex mesh. Additionally, SIS-IS has a process callback which alerts a process if any change occurs in the process table. This speeds the reaction to outage events.

---

<sup>1</sup> Since linked list was used as data structure, the search is linear ( $O(n)$ ). If a trie were used for the RIB, a more efficient search would have  $O(\log n)$  running time (see 4.3.1).

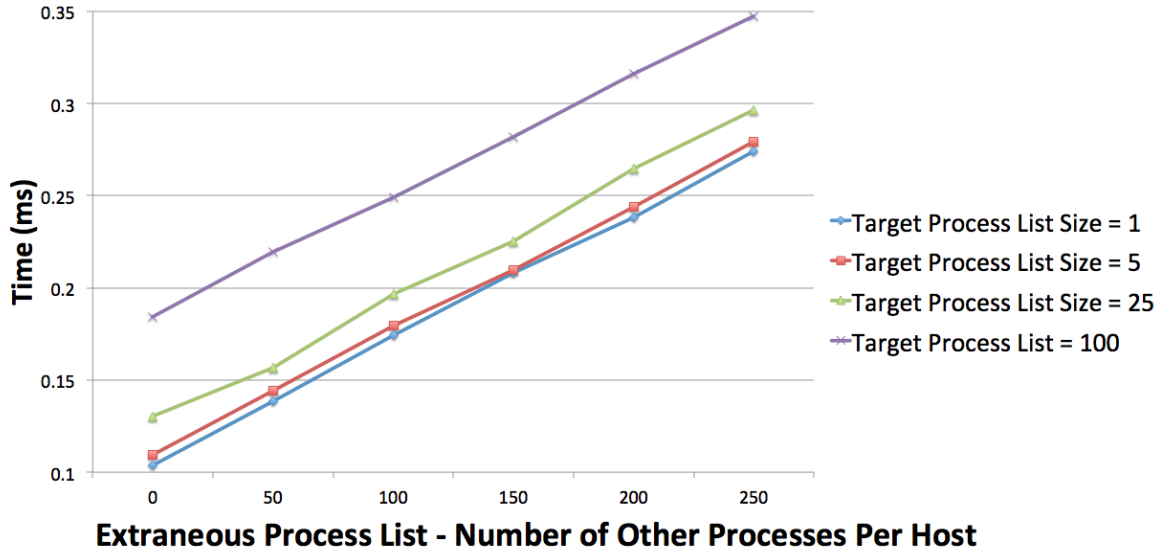


Figure 7.1: SIS-IS RIB - Average Time To Search

## 7.2 Implementation Overhead

In this Section we look at the drawbacks of our system, ie, how much slower is the system at installing an entry into the Routing Table over default router implementations. Initially, we measure the delay characteristics in installing a single RIB entry. From there, we perform a black-box test and measure the overall time delay for a router to switch paths after a failure.

### 7.2.1 Overhead in Installing a RIB Entry

In order to verify the scalability of the system, we need to make sure that running multiple replicas does not degrade performance and worsen scalability. To this end, in the first experiment we record the time delay for a Quagga non-HA OSPF process to install a routing table entry in FIB from its initial startup. Our experimental setup is as follows. The system is running Ubuntu Linux 12.04.5, with 512 MB in VMWare ESXi 5.1 environment. We run the experiment 30 times and average the results.

We then record the time delay for HA OSPF process groups to install a routing table entry in FIB. We run the process group in three-node, five-node, and nine-node configuration, respectively. The results are shown in Figure 7.3, where each non-HA

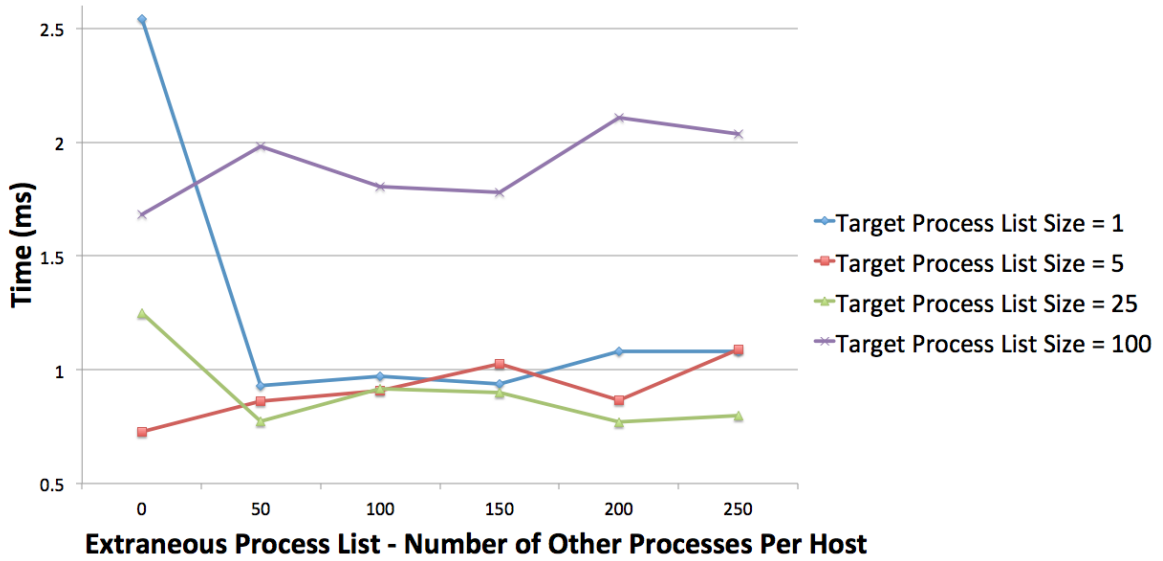


Figure 7.2: DHT - Average Time To Search

configuration is shown as a percent increase in time over the non-HA OSPF configuration.

The results show that adding replicas to a system does incur a time delay, however it is only 1.5 % in the worst-case for a three-node configuration. Adding additional replicas makes the time delay approach the delay of the non-HA process, where the nine-node configuration incurs only 0.6 % delay. We believe that this is due to the behavior of the voter, described in 4.2. Since the replicas are not synchronized to run in lock-step together, the voter needs to wait for a sibling majority before it can send output. Thus, the more siblings there are, the more messages are going to arrive at the voter for the same time period, and the bigger the likelihood that a majority will be reached. Overall, the extraneous time delays introduced by running multi-replica configurations are negligible.

### 7.2.2 Overhead Over Default Configuration

Using the same Virtual Machine and ESXi as described in previous section, we measure the overhead of our implementation over default Quagga and Cisco Router instances.

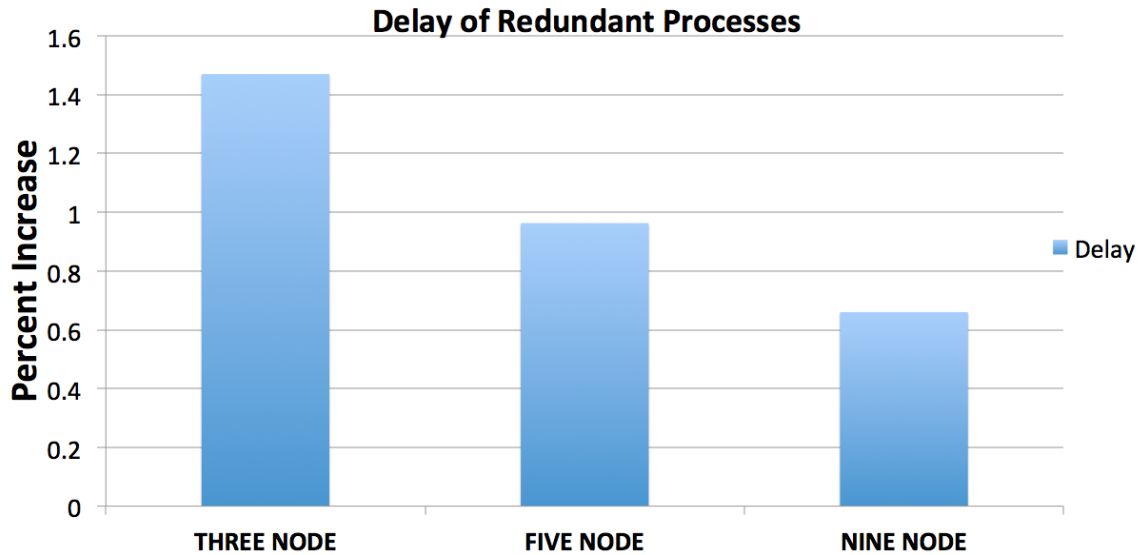


Figure 7.3: Percent Increase

We assume that we have the topology, as shown in Figure 7.4, where there is a host, N1, connected to a default router, called R1. R1 is connected to N2 through two redundant links, via routers R2 and R3. N1 is sending packets to N2, and they are going through R1 and R2. At a certain point, the router R2 crashes and becomes non-operational or its associated link fails. We look at how long it takes for R1 to react to topology change and switch its forwarding path from R2 to R3.

We look at a scenario where R1 is operational with no faults and switches the forwarding path right away. In this experiment we test the overall overhead of our system when everything is behaving normally. We tested the scenario delay characteristics using three separate implementations:

- A Cisco GNS3 Image - Our initial test uses a GNS3[2] architecture, which is shown in Figure 7.5. We use the Cisco 2600 Telco MZ 124 IOS image.
- A Default Non-HA Quagga Installation - Quagga, with network topology as shown in Figure 7.4, which is tested with OSPFv6, an implementation of OSPF with IPv6.
- The experimental HARRY implementation - finally, we test HARRY with three redundant siblings in a process group, with the same network topology shown in Figure 7.4.

The results are shown in Table 7.5, where the time to switch the path is measured for each separate implementation. All implementations take no more than 40 seconds,

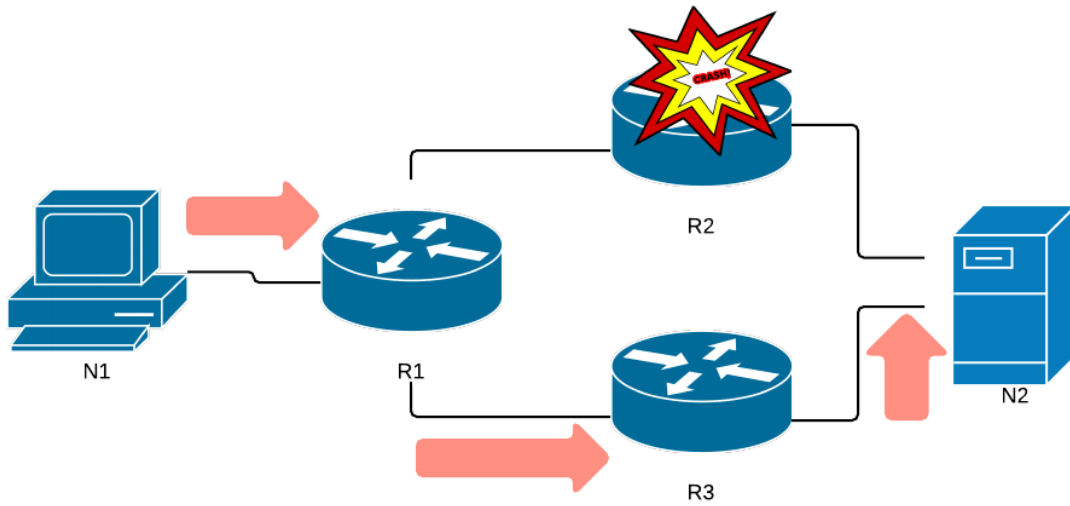


Figure 7.4: R1 Router Switching

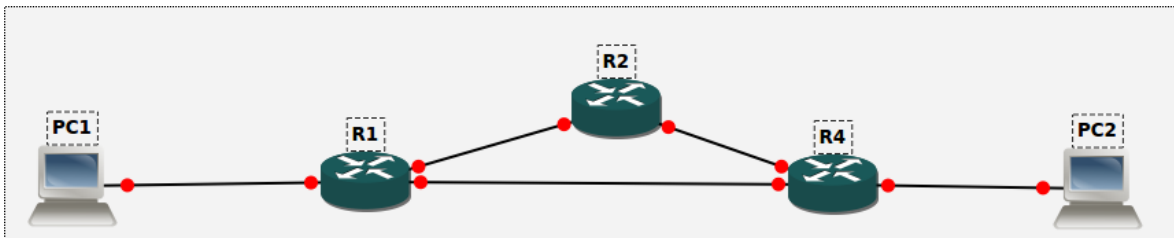


Figure 7.5: GNS3 Architecture

which is the amount in seconds that OSPF Dead Interval is set to. The reasoning is that it should be 3 times more than the hello interval, which is set to 10 seconds, since a packet could just be lost and then the network would make a drastic change. Both time delays in GNS3 and Quagga are almost equivalent. However, HARRY takes slightly longer for path switching, a 23 % increase. In HARRY, there are more functional components and therefore the message delay is slightly higher.

Please note that in all our experiments we interconnect Routers with Ethernet

Table 7.5: Path Switching (in sec)

GNS3 Cisco	Quagga	HARRY
31.3	32.2	39.7

link layer paths, and thus we need to wait on Application level timers in order to detect an outage. Outside of a network lab environment, routers are usually interconnected using more efficient synchronous transports such as SONET/SDH[69]. These networks can detect a link outage in microseconds and propagate that fault to the upper layers before a timeout expires.

### 7.3 Recovery Time

For the following test, we look at the recovery time of a process sibling, and the amount of faults our system can tolerate.

#### 7.3.1 Recovery Time of Single Sibling from Single Fault

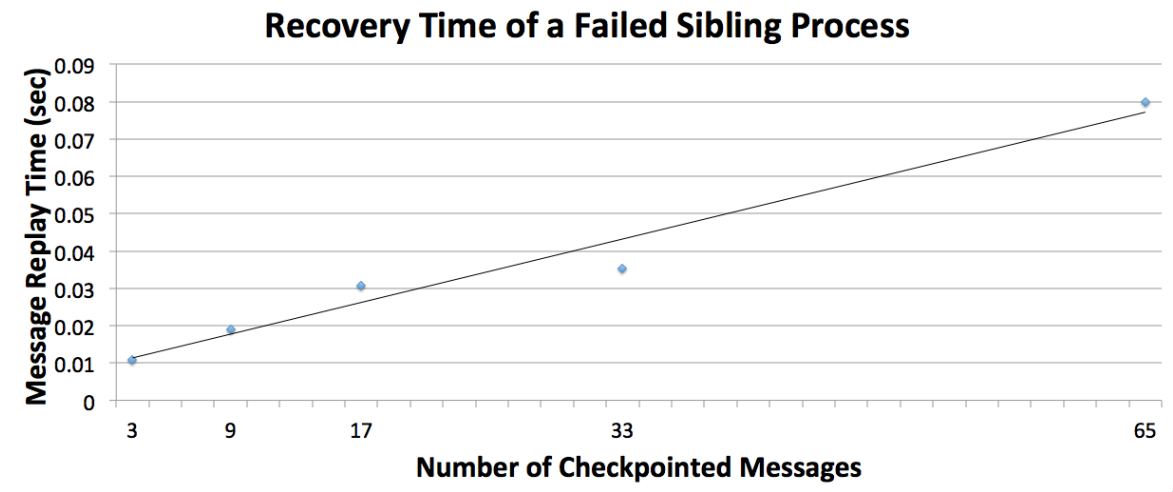


Figure 7.6: Time to ReSync

In this experiment we characterize the recovery time of a failed sibling process. Recovery is dominated by the State Replay Steps 3 and 4 described in Section 6.2.4.1.2, where checkpointed state messages are processed by the restarting sibling. We measure the time taken for a sibling to replay its previous messages. As seen in Figure 7.6, the recovery is linear in the number of items and is about 1 msec per item due to the transaction time of the state storage (DHT). An improved schema and query which returns multiple items simultaneously may improve performance.

### 7.3.2 Smallest Possible Fault Interarrival Period

The following experiment is a continuation of Section 7.3.1 where we measure the responsiveness of a Process Group. We start up the system and an adjacent OSPF6 neighbor. We want to find the smallest fault rate that our system can tolerate. A voter masks faults in the system, thus preserving the operability of the system. By masking faults, the adjacent neighbor should not be aware of any faults occurring in the system. As soon as the neighbor discovers that faults are occurring in the neighbor due to timeout of message delivery, our system is no longer tolerant to that fault rate.

As long as a sibling can stay up long enough to be able to send out a message, the voter is able to process that message successfully. We are interested in finding the smallest possible *fault inter arrival period* which would still enable the sibling to send out a message. As described in Section 6.2.3 and 6.2.4.1.2, a sibling needs a period of time in order to replay its state and synchronize itself with others. From a usability perspective of an external entity, only when the sibling starts interacting with external processes by sending messages progress can occur.

We kill all the siblings, and find the minimum time at which the system becomes non-operational. We keep the *fault inter arrival time* set at 10 seconds. When we kill a sibling, we measure the time taken from its termination until it actually sends its first message. If a fault were to occur before the first message is sent out, then the process is stuck in a cycle where it is constantly restarting but cannot actually send any messages. This observation leads us to conclude that the minimum fault rate that a process group can tolerate is the time from its termination until it sends its first message.

Our results, as shown in Figure 7.7, correspond to the time to ReSync, shown in Figure 7.6. The *fault inter arrival time* is affected by the number of checkpointed messages. Our results show that even at 65 checkpointed messages, we could tolerate a *fault inter arrival period* of 1 second. If we were to implement Checkpoint Collapsing, as covered in Section 8.3, then Recovery Time would be constant in regard to the number of Checkpointed Messages, and a guarantee of Non-Stop Operation that tolerates 1

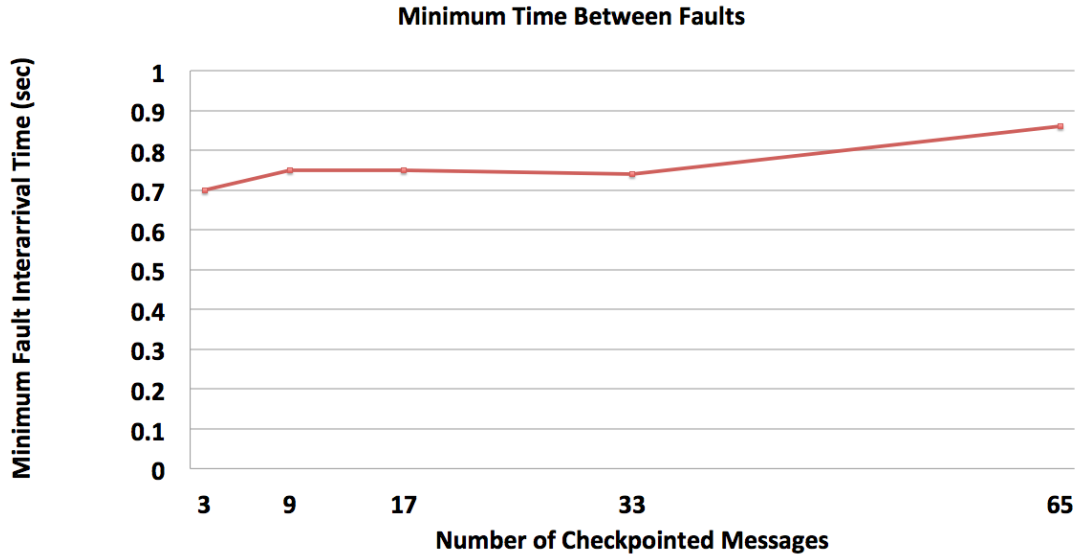


Figure 7.7: Minimum Fault Interarrival Time

second *fault inter arrival period* is feasible.

#### 7.4 Measuring Responsiveness (using ALL or ANY voter)

Different voting strategies, as described in Section 4.2.2, can vary significantly the performance characteristics of HARRY. In this section, we experimentally measure different voting strategies. We also vary the amount of siblings in a process group. Here, we keep the number of checkpoints in the DHT constant and do not vary them across each run.

The way we perform the experiment is as follows. We are interested in the time elapsed from sending an initial message until it is voted on successfully in the Controller and sent to the external system. We start a process group and record the time measured at which each sibling sends its first message. At the Controller, we also record when the first group of messages is successfully voted on. We find the time taken by subtracting the timestamp of earliest message sent from the timestamp when it was forwarded by the Controller. We performed the measurement 30 times and averaged the results.

The results are shown in Table 7.6, where VS is the Voting Strategy and the

Table 7.6: Different Voting Strategies (in msec)

VS	3-node	5-node	9-node
any	36.3	34.8	32.5
all	36.8	35.3	33.4

message voting delay is shown in milliseconds. As expected, voting with **ALL** has almost the same delay as voting with **ANY**, except that it is slightly longer. This is expected, since **ALL** will always take at least as long as **ANY**. Since **ALL** needs to wait until messages are received from all Siblings, it needs to wait until all Siblings have sent a message, something that **ANY** does not need to do. In these cases, and as confirmed through our results, the message delay is slightly higher for **ALL**. Depending on performance characteristics required by operator, an 0.5 or 0.9 increase in performance might warrant using **ANY** voting strategy, as long as the reliability characteristics of each voting strategy is well understood.

## Chapter 8

### CONCLUSION

In this work, we explored an HA model and applied it in the context of routing systems. The benefit of the work, however, is that it may be applied to any domain where HA is a desired property, including cloud computing and web content serving frameworks.

As outlined in [115], the "five 9s" reliability requirement - implying availability asymptotically approaching 100 % - is necessary for critical basic infrastructure. The goal of this project is to achieve similar level of highly reliable core router functionality. We are not the first to attempt to achieve these goals. In fact, these requirements[37] are still present and now even more stringent in current global service providers' desired feature sets[91]. There are many contributing factors currently preventing companies from implementing HA systems in a Routing Environment. The biggest factor is cost of implementing a new system, and migration from the old system to the new. By presenting an experimental system which satisfies these requirements, we outline steps necessary to implement an HA system, which can be used as an implementation model for future systems.

#### 8.1 Contributions

This research provides a number of contributions in the field of Router Reliability Research. Mainly, our work presents a mechanism for transforming a single process service such as a Routing Protocol, and modifying the software model so that the service supports Non-Stop operation. With the ever-growing ubiquity and availability of computing devices, such as tablets, smartphones, and other hand-held units, more stress is put on backbone infrastructure to handle simultaneous requests at any

time. Thus, even though our work is specifically focused on Routing, it may also be applied to any use-case where Non-Stop Real-Time Response is required. We cover the mechanisms necessary and suggest possible solutions to realizing said mechanisms.

One of the contributions of the work is a prototype, HARRY, of a distributed multi-line-card router that is able to forward customer's data to appropriate destinations as well as process control traffic which modifies FIB entries. The prototype achieves the goal of NSR (see Section 1.5), where multiple faults at the router do not impede the router's capabilities. Another unique contribution of our work is the use of Link-State Routing as general distributed database, a necessary requirement of HA distributed system environment.

The feasibility of our system is tested through various experiments that stress the robustness and capabilities of the system. We evaluate the distributed nature of HARRY and compare it to classical single line-card architecture. We verify the capability of NSR by introducing multiple faults into the system while the system is operational and observe its behavior.

## 8.2 Lessons Learned

Throughout the development of this work we gained new insights into building Distributed Applications, where state needs to be shared among multiple processes. We document our observations here for the benefit of people looking to implement a Distributed System. The key observations we had were as follows:

- Sharing State in a Distributed System is Hard - unlike a simplistic process that needs to only manage its own state, a sibling process belonging to a process group, which runs in an HA model such as N-Modular Redundancy, and needs to synchronize its state at certain points. There are many ways of doing that, and each method has strange corner cases and undefined behaviors. Generally, it is best to use strong consistency for small portions of the state that are time-critical, and for everything else to use eventual consistency.
- Conventional Methods of Debugging in a Distributed Application Won't Work - Using a debugger in a distributed application is not feasible, since hitting a breakpoint stops the execution of a process, while its siblings continue execution. Logging an application is the only feasible alternative, however sometimes it is difficult to predict exactly what to log. Additionally, inserting print statements everywhere is cumbersome and messy. Advanced debugging techniques such as

conditional breakpoints and watchpoints are sometimes necessary to gain insight into a bug.

- Getting Results in a Realistic Environment is Much Harder than in a Simulated One - before embarking on this research we mostly used a network simulator. In a network simulator it was much easier to generate results from experiments. In a real-time environment, shell scripts need to be written that orchestrate the whole system to perform specific actions, and different tools need to be written in order to parse the data. However, the benefit is that it is possible to implement almost anything and transitioning from a prototyping to production environment is trivial. Additionally, network simulators often have limitations.
- Build Distributed Systems out of Inherently Fault Tolerant Sub-Systems - a distributed system is inherently composed of multiple components that interact and work together to achieve a common goal. A well designed sub-component such as Distributed Key-Value Store (4.5) has the desired effect of providing fault-tolerant composable parts which can be combined in seamless fashion to build basic building blocks (see 4) that can be reused throughout the Distributed System. Since the operation of Distributed System is dependent on the health of its subcomponents, it is essential that the subcomponents are fault-tolerant.

### 8.3 Future Work

Only the most important critical pieces of our system were implemented. Many features were left out, due to lack of time and resources. For instance, even though we implemented most parts of OSPF, many features were left out. There is code to react and forward BGP traffic as well, however at this point BGP process groups have not yet been implemented. The rest of this section describes additional components that we would like to implement and explore if more time is available.

#### 8.3.1 Checkpoint Collapsing

As discussed in Section 4.4.1, different ways exist of checkpointing data. We chose to use message-based checkpointing, since when we were getting started it was easier to figure out exactly what to checkpoint. With state-based checkpointing, it is harder to isolate the data structures that are critical to checkpoint, especially if the data structures are still unknown. However, now that HARRY is more mature, we could improve on its checkpointing technique. In addition to having message-based

checkpointing, we could have a reaper process that activates at periodic intervals (in a similar fashion to how garbage collection activates[135]) and walks the checkpointed messages, thereby condensing them into a data structure that represents a state of a process. From there, a data structure representing process state exists, along with newly arrived messages. Periodically, these new messages would be replayed against the current state and integrated with it. Even though this algorithm adds complexity, it can potentially shrink down the message storage requirements. Additionally, a restarting process no longer needs to replay its state from messages and can load its state right away.

### 8.3.2 Transparent HA framework

The current limitation of the work is that an operator needs to modify the software structure of a process in order to make it redundant and integrate it with our system. A framework based off [81] covered in Section 3.5 that transparently acts as a hypervisor can be combined with our work. A hypervisor based system (or an enhanced version of something like GNS3[2]) could potentially extend this system to allow for the incorporation of alternate code sets to ease the implementation of n-version systems (discussed in Section 3.5).

### 8.3.3 Alternative Voting Implementations

As described in Section 4.2.1, Edit Distance or Context-Aware Comparison can be used as an alternative voting strategy. Various other strategies using Tree Edit Distance[22] and N-Gram[84] analysis can be also applied.

We implemented a simplistic voter which uses Levenshtein Distance as a basis for its comparison, however at the time of implementation uses oversimplified message format. It would be interesting to port the code to HARRY, and using a BGP traffic injection script[5], advertise fault-induced BGP RouteViews traffic[106] into HARRY. From there, a context-aware machine learning approach can be further explored.

## 8.4 Final Remarks

In this dissertation we demonstrate how to create Reliable Highly Available System by combining and modifying non-redundant code with several components which enhance reliability. The goal of this work is not to be too disruptive to a network, where an overhaul is required. Instead, an element itself could be made more reliable by running it in an HA environment. This work explores the environment, with a particular focus on Core Routers, and goes through the steps necessary to convert a simplistic process into N-Modular Redundant Process Group.

## BIBLIOGRAPHY

- [1] Etc monitoring concepts. <https://www.iav.com/en/publications/technical-publications/etc-monitoring-concepts>.
- [2] Gns3: Graphical network simulator.
- [3] Marathon: A container orchestration platform for mesos and dcos.
- [4] J. Cieslak D. Efimov P. Goupil A. Zolghadri, D. Henry. *Fault Diagnosis and Fault-Tolerant Control and Guidance for Aerospace Vehicles*. Springer, 2014.
- [5] Bahaa Al-Musawi, Philip Branch, and Grenville Armitage. BGP Replay Tool (BRT) v0.1. Technical Report 160304A, Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, 04 March 2016.
- [6] Alcatel-Lucent. Xrs ip core router: Big, bold, and booming. [XRSIPCoreRouter: Big, bold, and booming](#).
- [7] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.
- [8] H. Asenov and C. Cotton. Next generation resilient redundant router. In *2015 IEEE 15th International Conference on High Performance Switching and Routing (HPSR)*, July 2015.
- [9] H. Asenov, S. Sigwart, and C. Cotton. A routing based resource registry. In *Communications and Networking (BlackSeaCom), 2015 IEEE International Black Sea Conference on*, pages 176–180, May 2015.
- [10] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.
- [11] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, 11(12):1491, 1985.
- [12] James Aweya. Ip router architectures: an overview. *International Journal of Communication Systems*, 14(5):447–475, 2001.

- [13] Peter Bailis. Bridging the gap: Opportunities in coordination-avoiding databases. <http://www.bailis.org/blog/bridging-the-gap-opportunities-in-coordination-avoiding-databases/>.
- [14] Peter Bailis. *Coordination Avoidance in Distributed Databases*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2015.
- [15] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.
- [16] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '03*, pages 170–177, New York, NY, USA, 2003. ACM.
- [17] Basho. Strong consistency. <https://docs.basho.com/riak/kv/2.1.4/developing/app-guide/strong-consistency/>.
- [18] Basho. Introducing riak 2.0: Data types, strong consistency, full-text search, and much more. <http://basho.com/posts/technical/introducing-riak-2-0/>, October 2013.
- [19] Eric Bauer and Randee Adams. *Reliability and Availability of Cloud Computing*. Wiley-IEEE Press, 1st edition, 2012.
- [20] Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. March 2014.
- [21] Randy Bias. Pets vs. cattle: The elastic cloud story. <http://cloudscaling.com/blog/cloud-computing/pets-vs-cattle-the-elastic-cloud-story/>.
- [22] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, June 2005.
- [23] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.
- [24] Kenneth P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, December 1993.
- [25] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.
- [26] Kenneth P. Birman and Robert V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

- [27] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb 2012.
- [28] Aaron Brown and David A Patterson. Embracing failure: A case for recovery-oriented computing (roc). In *High Performance Transaction Processing Symposium*, volume 10, pages 3–8, 2001.
- [29] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Distributed systems (2nd ed.). chapter The Primary-backup Approach, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.
- [30] G. Buja, S. Castellan, R. Menis, and A. Zuccollo. Dependability of safety-critical systems. In *Industrial Technology, 2004. IEEE ICIT '04. 2004 IEEE International Conference on*, volume 3, pages 1561–1566 Vol. 3, Dec 2004.
- [31] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.
- [32] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.
- [33] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.
- [34] H.J. Chao. Next generation routers. *Proceedings of the IEEE*, 90(9):1518–1558, Sep 2002.
- [35] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the v kernel. *ACM Trans. Comput. Syst.*, 3(2):77–107, May 1985.
- [36] J. Cornwell and A. Kongmunvattana. Efficient system-level remote checkpointing technique for blcr. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 1002–1007, April 2011.
- [37] Chase Cotton. The key problems we face building core routers. <http://www.cs.vu.nl/~kielmann/asfr09/cotton-provider.pdf>, October 2009.
- [38] A. Csaszar, G. Enyedi, G. Enyedi, G. Retvari, and M. Hidell. Converging the evolution of router architectures and ip networks. *IEEE Network*, 21(4):8–14, July 2007.
- [39] E. G. Sirer D. Altinbuken. Coordination service for distributed systems.

- [40] A. Das, I. Gupta, and A. Motivala. Swim: scalable weakly-consistent infection-style process group membership protocol. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 303–312, 2002.
- [41] Jean Dean. Software engineering advice from building large-scale distributed systems. <http://static.googleusercontent.com/media/research.google.com/en/us/people/jeff/stanford-295-talk.pdf>.
- [42] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.
- [43] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.
- [44] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC ’87*, pages 1–12, New York, NY, USA, 1987. ACM.
- [45] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP ’09*, pages 15–28, New York, NY, USA, 2009. ACM.
- [46] P. Domingues, F. Araujo, and L. M. Silva. A dht-based infrastructure for sharing checkpoints in desktop grid computing. In *2006 Second IEEE International Conference on e-Science and Grid Computing (e-Science’06)*, pages 76–76, Dec 2006.
- [47] Jeff Doyle. Nsf, nsr, and gr. <http://www.networkworld.com/article/2347847/cisco-subnet/nsf--nsr--and-gr.html>, June 2007.
- [48] Wenbo Duan, Limin Xiao, Deguo Li, Yuanhao Zhou, Rui Liu, Li Ruan, Yinben Xia, and Mingming Zhu. Ofbgp: A scalable, highly available bgp architecture for sdn. In *Mobile Ad Hoc and Sensor Systems (MASS), 2014 IEEE 11th International Conference on*, pages 557–562, Oct 2014.
- [49] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014.

- [50] A. Fekih. Fault diagnosis and fault tolerant control design for aerospace systems: A bibliographical review. In *American Control Conference (ACC), 2014*, pages 1286–1291, June 2014.
- [51] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.
- [52] Open Networking Foundation. Software-defined networking: The new norm for networks. Technical report, April 2012.
- [53] Open Networking Foundation. Openflow switch specification. Technical report, October 2013.
- [54] Rachit Garg and Praveen Kumar. Article:a review of fault tolerant checkpointing protocols for mobile computing systems. *International Journal of Computer Applications*, 3(2):8–19, June 2010. Published By Foundation of Computer Science.
- [55] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.
- [56] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, Apr 1997.
- [57] O. Hagsand, M. Hidell, and P. Sjodin. Design and implementation of a distributed router. In *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, pages 227–232, Dec 2005.
- [58] W. Hamzeh and A. Hafid. A scalable cluster distributed bgp architecture for next generation routers. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, pages 161–168, Oct 2009.
- [59] Wissam Hamzeh. *A parallel distributed architecture and tolerant breakdowns for protocol cross-domain BGP in the heart of the Internet*. PhD thesis, University of Montreal, Montreal, Canada, December 2010.
- [60] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing extensible ip router software. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2, NSDI’05*, pages 189–202, Berkeley, CA, USA, 2005. USENIX Association.
- [61] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.
- [62] HashiCorp. Consensus protocol.

- [63] HashiCorp. Serf: Gossip protocol.
- [64] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
- [65] M. Hidell, P. Sjodin, and O. Hagsand. Reliable multicast for control in distributed routers. In *High Performance Switching and Routing, 2005. HPSR. 2005 Workshop on*, pages 133–137, May 2005.
- [66] Markus Hidell. *Decentralized Modular Router Architecture*. PhD thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2006.
- [67] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11*, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.
- [68] Jr. Hopkins, A.L., III Smith, T.B., and J.H. Lala. Ftmp: A highly reliable fault-tolerant multiprocess for aircraft. *Proceedings of the IEEE*, 66(10):1221–1239, Oct 1978.
- [69] Ray Horak. *Telecommunications and Data Communications Handbook*. Number p. 476. Wiley-Interscience, 2007.
- [70] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10*, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.
- [71] Cisco Inc. Cisco crs carrier routing system 16-slot line chassis system description.
- [72] Cisco Inc. Cisco ios xr routing configuration guide for the cisco crs router.
- [73] Cisco Inc. Requirements for next-generation core routing systems.
- [74] Juniper Inc. T-series core routers datasheet.
- [75] Rolf Isermann, R. Schwarz, and S. Stolzl. Fault-tolerant drive-by-wire systems. *Control Systems, IEEE*, 22(5):64–81, Oct 2002.
- [76] Rolf Isermann, R. Schwarz, and S. Stolzl. Fault-tolerant drive-by-wire systems. *Control Systems, IEEE*, 22(5):64–81, Oct 2002.
- [77] M. Schoffstall J. Davis J. Case, M. Fedor. A simple network management protocol (snmp). Rfc, IETF, 1990.

- [78] P. Jakma and D. Lamparter. Introduction to the quagga routing suite. *Network, IEEE*, 28(2):42–48, March 2014.
- [79] Shanley Kane. Why your riak cluster should have at least five nodes, April 2012.
- [80] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.
- [81] Eric Keller, Minlan Yu, Matthew Caesar, and Jennifer Rexford. Virtually eliminating router bugs. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 13–24, New York, NY, USA, 2009. ACM.
- [82] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.
- [83] A. Kohn, M. Kasmeyer, R. Schneider, A. Roger, C. Stellwag, and A. Herkersdorf. Fail-operational in safety-related automotive multi-core systems. In *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*, pages 1–4, June 2015.
- [84] Grzegorz Kondrak. N-gram similarity and distance. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval*, SPIRE'05, pages 115–126, Berlin, Heidelberg, 2005. Springer-Verlag.
- [85] Phillip Koopman. Safety requires no single points of failure. <http://betterembsw.blogspot.com/2014/03/safety-requires-no-single-points-of.html>.
- [86] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
- [87] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, December 2001.
- [88] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.
- [89] Butler W. Lampson. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, pages 1–17, London, UK, UK, 1996. Springer-Verlag.

- [90] G. Latif-Shabgahi, J.M. Bass, and S. Bennett. A taxonomy for software voting algorithms used in safety-critical systems. *Reliability, IEEE Transactions on*, 53(3):319–328, Sept 2004.
- [91] Peter Lothberg. Terastream - a simplified service delivery network.
- [92] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys and Tutorials, IEEE*, 7(2):72–93, Second 2005.
- [93] Parisa Jalili Marandi, Samuel Benz, Fernando Pedonea, and Kenneth P. Birman. The performance of paxos in the cloud. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems, SRDS '14*, pages 41–50, Washington, DC, USA, 2014. IEEE Computer Society.
- [94] Tudor Marian, Ki Suh Lee, and Hakim Weatherspoon. Netslices: Scalable multi-core packet processing in user-space. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '12*, pages 27–38, New York, NY, USA, 2012. ACM.
- [95] David Mazieres. Paxos made practical. <http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf>.
- [96] Caitie McCaffrey. Making the case for building scalable stateful services in the modern era. <http://highscalability.com/blog/2015/10/12/making-the-case-for-building-scalable-stateful-services-in-t.html>.
- [97] Christopher Meiklejohn. Riak pg: Distributed process groups on dynamo-style distributed storage. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang, Erlang '13*, pages 27–32, New York, NY, USA, 2013. ACM.
- [98] K. Mohror, A. Moody, and B. R. de Supinski. Asynchronous checkpoint migration with mrnet in the scalable checkpoint / restart library. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, June 2012.
- [99] John T. Moy. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
- [100] NASA. Shuttle reference manual. [spaceflight.nasa.gov/shuttle/reference/shutref/orbiter/avionics/dps/gpc.html](http://spaceflight.nasa.gov/shuttle/reference/shutref/orbiter/avionics/dps/gpc.html).
- [101] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001.
- [102] Pablo Neira-Ayuso, Rafael M. Gasca, and Laurent Lefevre. Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience*, 40(9):797–810, 2010.

- [103] K. K. Nguyen, B. Jaumard, and A. Agarwal. A distributed and scalable routing table manager for the next generation of ip routers. *IEEE Network*, 22(2):6–14, March 2008.
- [104] K. K. Nguyen, H. Mahkoun, B. Jaumard, C. Assi, and M. Lanoue. Toward a distributed control plane architecture for next generation routers. In *Universal Multiservice Networks, 2007. ECUMN '07. Fourth European Conference on*, pages 173–182, Feb 2007.
- [105] P Glenn Norman. The new ap101s general-purpose computer (gpc) for the space shuttle. *Proceedings of the IEEE*, 75(3):308–319, 1987.
- [106] University of Oregon. Route views archive project. <http://bgplay.routeviews.org/>.
- [107] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.
- [108] OPENDAYLIGHT. Opendaylight.
- [109] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.
- [110] Ivan Pepelnjak. Process, fast and cef switching and packet punting. <http://blog.ipspace.net/2013/02/process-fast-and-cef-switching-and.html>, Feb 2013.
- [111] B. Phillips. Using etcd with coreos.
- [112] P.K. Reddy and M. Kitsuregawa. Reducing the blocking in two-phase commit protocol employing backup sites. In *Cooperative Information Systems, 1998. Proceedings. 3rd IFCIS International Conference on*, pages 406–415, Aug 1998.
- [113] Riak. Active anti-entropy.
- [114] Luigi Rizzo. Revisiting network i/o apis: The netmap framework. *Queue*, 10(1):30:30–30:39, January 2012.
- [115] Ken Birman Douglas Comer Chase Cotton Thilo Kielmann Bill Lehr Robert VanRenesse Robert Surton Jonathan M. Smith Robert M. Broberg, Andrei Agapi. Clouds, cable and connectivity: Future internets and router requirements. In *In Proc. Cable Connection Spring Technical Conference*, 2011.

- [116] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.
- [117] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.
- [118] Stephen Sigwart. System is-is : a routing based resource registry to support distributed routers. Master’s thesis, University of Delaware, 2012.
- [119] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’81, pages 133–142, New York, NY, USA, 1981. ACM.
- [120] J.R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, Jan 1976.
- [121] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC ’06, New York, NY, USA, 2006. ACM.
- [122] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1):17–32, Feb 2003.
- [123] R. Surton, K. Birman, and R. van Renesse. Application-driven tcp recovery and non-stop bgp. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12, June 2013.
- [124] A. Svoboda. From mechanical linkages to electronic computers: Recollections from czechoslovakia,. *IEEE*, 1980.
- [125] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
- [126] Basho Technologies. Riak. <http://wiki.basho.com/Riak.html>.
- [127] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP ’95, pages 172–182, New York, NY, USA, 1995. ACM.
- [128] J. Tourrilhes, P. Sharma, S. Banerjee, and J. Pettit. Sdn and openflow evolution: A standards perspective. *Computer*, 47(11):22–29, Nov 2014.

- [129] Brian Troutwine. Let it crash! the erlang approach to building reliable services. <http://www.infoq.com/presentations/erlang-reliable-services>.
- [130] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015.
- [131] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, October 2008.
- [132] L. Wang, D. Massey, K. Patel, and L. Zhang. Frtr: a scalable mechanism for global routing table consistency. In *Dependable Systems and Networks, 2004 International Conference on*, pages 465–474, June 2004.
- [133] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, Oct 1978.
- [134] Wikipedia. Automotive safety integrity level — wikipedia, the free encyclopedia, 2015. [Online; accessed 10-May-2016].
- [135] Wikipedia. Garbage collection (computer science) — wikipedia, the free encyclopedia, 2016. [Online; accessed 25-April-2016].
- [136] Wikipedia. Google self-driving car — wikipedia, the free encyclopedia, 2016.
- [137] Wikipedia. Virtual synchrony — wikipedia, the free encyclopedia, 2016.
- [138] Wikipedia. Vsync (computing) — wikipedia, the free encyclopedia, 2016.
- [139] S. Hares Y. Rekhter, T. Li. A border gateway protocol 4 (bgp-4), January 2006. RFC 4271.
- [140] Xiaozhe Zhang, Xicheng Lu, Jinshu Su, Baosheng Wang, and Zexin Lu. Sdbgp: A scalable, distributed bgp routing protocol implementation. In *High Performance Switching and Routing (HPSR), 2011 IEEE 12th International Conference on*, pages 191–196, July 2011.
- [141] A. Zinin. *Cisco IP Routing: Packet Forwarding and Intra-domain Routing Protocols*. Addison-Wesley, 2002.

## Appendix

### A IMPLEMENTATION DETAILS

#### A.1 RouteFlow

The communications channel between all the processes in the System is called RouteFlow.

A RouteFlow (RF) message always starts with a **rfp\_header**, which is a structure with the following fields:

- Version - the version of RF protocol
- Type - the type of message we should expect following the header
- length - the length of the whole packet, including the length of the header
- Xid - transaction id associated with this packet. Transaction ids are used for message ordering and associating requests with replies

The **type** field dictates what the body of the packet will contain and how to interpret its contents. The rf type may be:

- Hello - generic message which every process sends after initial connection.
- Features Request - a request to retrieve the interface ports of the Datapath.
- Features Reply - a reply with the interface number, state, mtu and name prefilled for each interface on the corresponding Datapath.
- Redistribute Request - a request to redistribute the routes of the Controller to the Siblings.
- Route Add Reply- a response to the redistribute request message. The message contains 4 bytes of the prefix address along with the prefix length. A separate message is sent for each separate route.
- Stats Routes Request - a request to get the Routes from the Data Plane.

- Stats Routes Reply - a reply to the stats routes request message. A message containing all the routes is sent back to the Controller. The information that is sent back for a route is the prefix address, the prefix length, the route metric, the distance of the route, and the routing table it is set in.
- If Address Request - a request to get the address for each interface at the Controller.
- IPv4 Address Add Reply - a reply to a corresponding If Address Request.
- IPv6 Address Add Reply - a reply to a corresponding If Address Request.
- Addresses Request - a request to get the address for each interface at the Data-plane.
- Addresses Reply - a reply to a corresponding Address Request message. A message containing all the addresses, both IPv4 and IPv6, as well as their interface number, is sent back to the Controller.
- IPv6 Route Set Request - request to set an IPv6 Route in the DataPlane from the Sibling. Upon reception, the Controller simply forwards the message. The message contains a prefix, prefix length, the address of the nexthop and its corresponding interface index.
- Redistribute Leader Elect - a message from a Sibling to Controller, informing the Controller that the Leader has changed. The Controller keeps track of the leader sibling for process restart purposes.
- Replica Ex - a message sent during leader election from a sibling to its peers indicating replica ID and whether it thinks it is the sibling or not.
- Leader Elect - a message sent from Controller to Siblings indicating they are ready for leader election.

For all requests, their corresponding reply messages contain the same xid in the header, so that we are able to match up replies to requests and vice versa.

When the **type** field is set to either **FORWARD\_OSPF6** or **FORWARD\_BGP**, then either an OSPF or BGP message is encapsulated in the RF message. The Controller and Zebralite simply forward these messages without having to know how to interpret the data. A Sibling and Punter generates and processes these messages.

## A.2 Controller Initialization with Zebralite

Controller sends Hello, followed by Features Request and Stats Routes Request, as illustrated in Figure A.1. A Features Reply followed by Stats Routes Reply messages are received from Zebralite, which contain all the Interface information (excluding their addresses) and all the Routes of the DP Line Card. Only a single Stats Routes Reply message is sent back, containing all the Routes in a single message. After a Features Reply is parsed, the Controller sends an Addresses Request to retrieve all the addresses for their corresponding interfaces. A single Addresses Reply message is received, containing all the Address information of that particular DP Line Card.

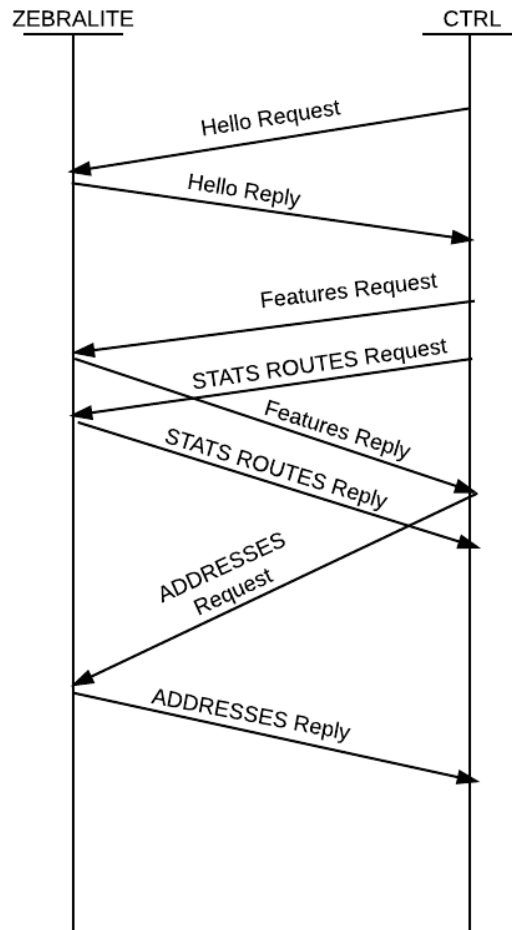


Figure A.1: Controller Initialization With Zebralite

### A.3 Sibling Initialization with Controllers

In a Sibling Process the object Controller (Ctrl) Client corresponds to a single client between the Controller and the Sibling. Since there are multiple Controllers for each Sibling, a list of ctrl clients corresponds to multiple simultaneous connections between a Sibling and Controllers. The list is managed by the object Sibling Ctrl.

Figure A.2 shows the initialization sequence between a Controller and a single Sibling only. Initially, a Hello Message Exchange makes sure that all communication endpoints are communicating properly. From there, the Sibling sends Features Request and Redistribute Request message in parallel. A Features Reply message contains a list of the names, index ids, mtu, etc., of the world interfaces that the Controller is managing. For a single Redistribute Request, multiple Routes Add Reply messages are received, where each message contains an individual entry in the world FIB of the Data Plane.

Once the Ctrl Client finds a name that corresponds to a world interface that it is managing from the Features Reply message, it issues an "interface up" event, updates its state and signals to the Sibling Ctrl that its state is updated (shown in Figure A.3). Sibling Ctrl additionally waits for an "interface up" event from all its Ctrl Client objects (shown in Figure A.4). Once all the Ctrl Clients are past "interface up", Sibling Ctrl changes its state to ALL\_INTERFACE\_UP.

A single If Addr Request is sent from a Sibling when it has processed and reacted fully to Features Reply. The Controller responds with multiple Address Add Reply messages, where each message contains an id and address of a world interface. When the Controller receives an If Addr Request from all the Siblings, a Leader Elect message is sent to all the Siblings.

The Ctrl Client might be either in RCVD\_HELLO or INTERFACE\_UP state when a Leader Elect message is received. It updates its state and signals to Sibling Ctrl for Leader Election to start. If the "interface up" event has already occurred, the state is transitioned to CONNECTED, and the initial exchange between the Controller and Siblings is complete. Once Sibling Ctrl has received all the Leader Election Start signals

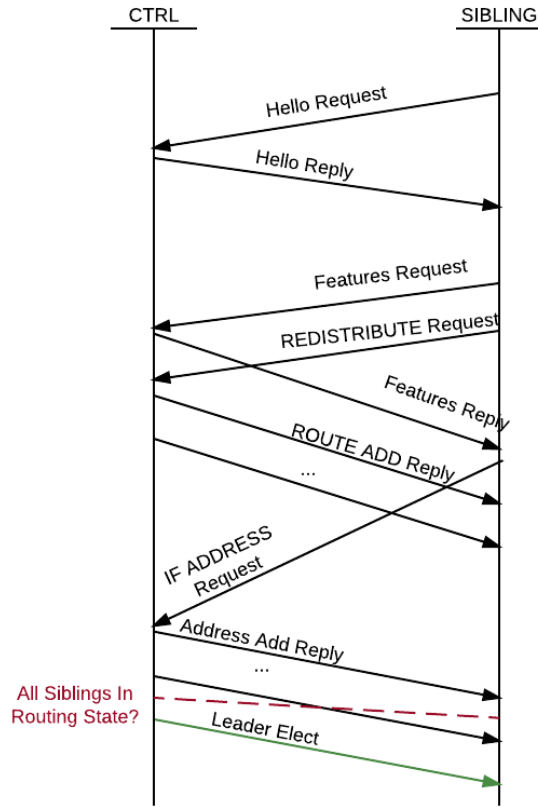


Figure A.2: Controller Initialization With Siblings

from each Ctrl Client object, Leader Election, as described in 6.2.2, is performed. When both all interfaces are up and Leader Election is complete, the actual Routing Protocol message exchange is performed, such as sending application-level hello messages (OSPF Hello, BGP Keepalive, etc.).

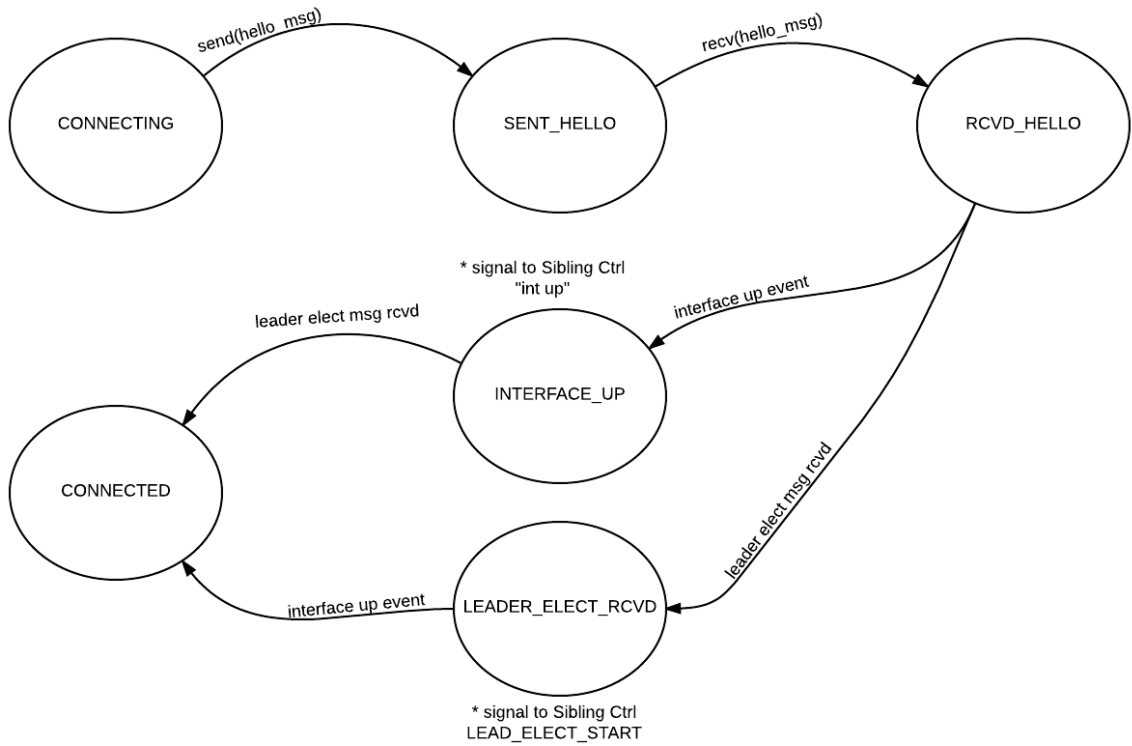


Figure A.3: Controller Client State Diagram

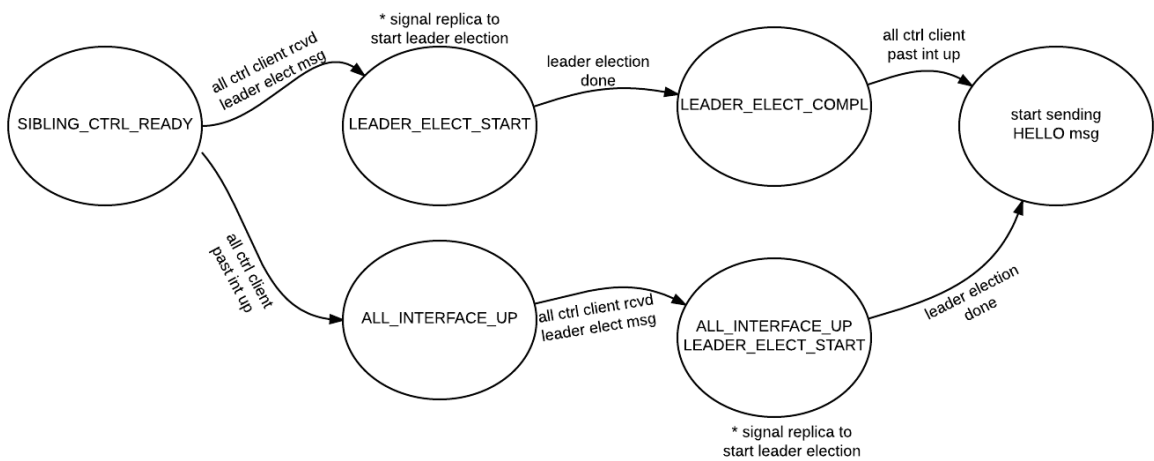


Figure A.4: Sibling Ctrl State Diagram at Siblings